SIAM J. COMPUT.
Vol. 19, No. 1, pp. 1-31, February 1990

AN OPTIMAL LINEAR-TIME PARALLEL PARSER FOR TREE ADJOINING LANGUAGES*

MICHAEL A. PALIS†, SUNIL SHENDE†, AND DAVID S. L. WEI†

Abstract. An optimal parallel recognition/parsing algorithm is presented for languages generated by tree adjoining grammars (TAGs), a grammatical system for natural language. TAGs are strictly more powerful than context-free grammars (CFGs), e.g., they can generate $\{a^n b^n c^n \mid n \geqq 0\}$, which is not context-free. However, serial parsing of TAGs is also slower, having time complexity $O(n^6)$ for inputs of length n (as opposed to $O(n^3)$ for CFGs). The parallel algorithm achieves optimal speedup: it runs in linear time on a five-dimensional array of n^5 processors. Moreover, the processors are finite-state; i.e., their function and size depends only on the underlying grammar and not on the length of the input.

Key words. language recognition and parsing, tree adjoining languages, context-free languages, mesh-connected processor arrays

AMS(MOS) subject classifications. 68Q80, 68Q35, 68Q45, 68Q50, 68S05

1. Introduction.

Language recognition and parsing are important problems that arise in many applications, e.g., compiler construction, natural language processing, and syntactic pattern recognition. Much of the work in this area has centered on context-free languages (CFLs) and its subclasses. Although many subclasses of CFLs can be parsed in linear time, the fastest known practical parsing algorithms for general CFLs (Cocke-Younger-Kasami's and Earley's algorithm) have time complexity $O(n^3)$ for inputs of length n [AHO72], [HOPC79]. An asymptotically faster algorithm that runs in $O(M(n))$ time has been given by Valiant [VALI75], where $M(n)$ is the time to multiply two $n \times n$ Boolean matrices. Currently, the best-known upper bound on $M(n)$ is $O(n^{2.376})$ [COPP87]. However, the constant of proportionality in Valiant's algorithm is too large for practical applications.

Recent research has sought to decrease the time bound for CFL recognition and parsing by introducing parallelism. The parallel recognition of CFLs was first considered by Kosaraju in [KOSA75], where he showed that CFLs can be recognized by two-dimensional arrays of finite-state machines in linear time. His construction is a parallelization of the Cocke-Younger-Kasami (CYK) dynamic programming algorithm for recognizing the strings generated by a context-free grammar in Chomsky normal form (CNF). Later, Chiang and Fu [CHIA84] extended this result to the parsing problem (i.e., if the string is in the language, output a parse tree of the string). Their algorithm that performs both recognition and parsing is a parallel implementation of Earley's algorithm (that does not constrain the grammar to be in CNF) and runs in linear time on a two-dimensional systolic array of $O(n^2)$ processors. Unfortunately, for the parsing phase of the algorithm, the processors are no longer finite-state because they store and manipulate log n-bit numbers. A fully finite-state linear-time parallel parser (based on the CYK algorithm) was later given by Chang, Ibarra, and Palis in [CHAN87].

* Received by the editors June 15, 1987; accepted for publication (in revised form) January 1, 1989. This research was partially supported by Army Research Office grant, DAA29-84-9-0027, National Science Foundation grants MSC-8219116-CER, MCS-82-07294, DCR-84-10413, MCS-83-05221, and Defense Advanced Research Projects Agency grant N00014-85-K-0018.

† Department of Computer and Information Science, University of Pennsylvania, Philadelphia, Pennsylvania 19104-6389.

In 1975, Joshi, Levy, and Takahashi [JOSH75] introduced a grammatical system called tree adjoining grammar (TAG) that is strictly more powerful (in terms of generative capacity) than context-free grammars. For example, TAGs can generate $\{a^n b^n c^n \mid n \geqq 0\}$, which is not context-free. Although initially studied for their mathematical properties, TAGs have recently been rediscovered as a good grammatical system for natural language [KROC85]. It was not until recently that it has been shown that tree adjoining languages (TALs) generated by TAGs are polynomial-time parsable [VIJA86]. However, the serial parsing algorithm is much more complicated and runs slower than that for CFLs, having time complexity $O(n^6)$ for inputs of length n [VIJA86].

In this paper, we present a parallel recognition and parsing algorithm for TALs. Our algorithm achieves optimal speedup: it runs in linear time on a five-dimensional array of n^5 processors. Moreover, the processors are finite-state, i.e., their function and size depend only on the underlying grammar and not on the length of the input string.

The paper is divided into five sections, in addition to this section. Section 2 briefly introduces TAGs and presents the serial parsing algorithm given in [VIJA86]. Section 3 discusses the array model. The parallel recognition algorithm is described in § 4, and its extension to parsing is discussed in § 5. Section 6 ends the paper with some concluding remarks.

2. Tree adjoining grammars. In this section, we define tree adjoining grammars and present the sequential recognition algorithm given in [VIJA86]. We also define what constitutes a parse tree of an input string and describe how it can be recovered by a simple extension to the recognition algorithm.

2.1. Definition of TAGs. Unlike context-free grammars that are defined in terms of rewriting rules on symbols over a finite alphabet, TAGs are defined in terms of an operation called *adjunction* on labeled trees. Formally , TAG is a 5-tuple $G = (N, \Sigma, I, A, S)$, where

N is a finite set of *nonterminal symbols*,

Σ is a finite set of *terminal symbols* disjoint from N,

I is a finite set of labeled *initial trees*,

A is a finite set of labeled *auxiliary trees*,

$S \in N$ is the distinguished *start symbol.*

Initial and auxiliary trees are called the *elementary trees* of the grammar. All internal nodes of elementary trees are labeled with nonterminal symbols. In addition, every initial tree is labeled at the root by the start symbol S and has leaf nodes labeled with symbols in $\Sigma \cup \{\varepsilon\}$ (where ε is the empty string). An auxiliary tree has both its root and exactly one leaf node (called the *foot node*) labeled with the *same* nonterminal symbol. All other leaf nodes are labeled with symbols in $\Sigma \cup \{\varepsilon\}$, at least one of which has a label strictly in Σ.

An operation called *adjunction* composes trees of the grammar as follows. Let γ be a tree containing some internal node labeled X, and let β be an auxiliary tree whose root is labeled with the same symbol X. (See Fig. 2.1 but ignore the C_i's for the moment.) Then *adjoining β into γ* at the node labeled X results in the composite tree α. Informally, the subtree t of γ rooted at the node labeled X is excised, β is inserted in its place, and t is attached to the unique foot node of β. The resulting tree is α.

In general, the formalism allows the possibility of *constrained adjunction* at a node, i.e., we can associate with every node a corresponding subset of auxiliary trees that can be adjoined at that node. This subset is denoted as the *constraint* associated

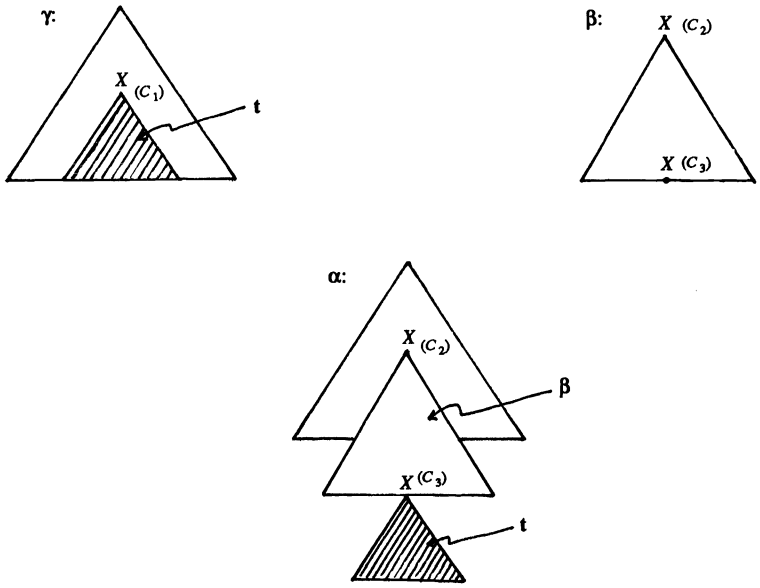


FIG. 2.1. *The adjunction operation.*

with the node; adjunction of an auxiliary tree at the node is allowed only if the node's constraint set contains the auxiliary tree. Constraints may be of two types: *selective* or *obligatory*. The former case corresponds to selectively adjoining zero or one of the auxiliary trees in the constraint set, whereas the latter corresponds to necessarily adjoining one of the trees from the set. Constraints are represented as tuples of the form $\langle type, subset \rangle$ where the type can take values "*SA*" or "*OA*" denoting selective or obligatory adjunction, respectively, from the specified set. In constrained adjunction, the constraint changes at the node where the adjunction took place, as indicated by the C_i's in Fig. 2.1. More precisely, this node gets the constraint of the root of the auxiliary tree that is adjointed at the node.

In the subsequence, we assume that elementary trees of the grammar are assigned unique tree numbers and that within a tree, nodes have unique positional indices. We adopt the convention that if Γ and Δ are in the same tree then *index* $(\Gamma) <$ *index* (Δ) if and only if a postorder traversal of the tree visits Γ before Δ. Thus, each node is represented as a tuple $\langle$*tree-number, index, label, constraint*$\rangle$. Node Δ is *adjoinable* at node Θ if and only if Δ is the root node of some auxiliary tree α, *label* (Δ) = *label* (Θ), *constraint* $(\Theta) = \langle type, S \rangle$, and $\alpha \in S$.

Tree α *elementary derives* tree β (denoted $\alpha \rightarrow \beta$) if and only if β results from α by adjoining an auxiliary tree at some node in α. α *derives* β (denoted $\alpha \rightarrow^* \beta$) if and only if there is a sequence of zero or more trees starting with α and ending in β such that every tree in the sequence elementarily derives its successor. β is called a *derived* tree if and only if $\alpha \rightarrow^* \beta$ for some elementary tree α; in particular, if α is an initial (auxiliary) tree, then β is called a derived initial (auxiliary) tree. The *frontier* of a tree is defined as the left-to-right ordered sequence of leaf nodes of the tree. The *yield* of the tree is the corresponding string of labels of the frontier nodes. It can be verified that every initial tree of a TAG derives trees whose yields are strings of terminal symbols. Accordingly, the *tree adjoining language* (TAL) $L(G)$ generated by a TAG G is defined as follows:

$L(G) = \{w \in \Sigma^* \mid w$ is the yield of a derived initial tree that does not contain any nodes with constraints of type "*OA*"$\}$.

For example, consider the TAG $G = (\{S\}, \{a, b, c\}, \alpha, \beta, S)$ shown in Fig. 2.2(a). If the auxiliary tree β is adjoined into the initial tree α at its root node, the derived tree γ_1 results (see Fig. 2.2(b)). Adjoining β into γ_1 at the node indicated by the arrow produces a new derived tree γ_2. This process can be continued producing larger derived trees. It can be shown that all such derived trees have yields of the form $a^i b^i c^i$. Moreover, the trees have no nodes with constraints of type "*OA*." Thus, $L(G) = \{a^n b^n c^n \mid n \geqq 0\}$. Note that $L(G)$ is not context-free.

The sequential algorithm described in the next section makes certain assumptions about the structure of the tree adjoining grammar; in particular, it is assumed that the TAG is in *normal form.* A TAG is in normal form if and only if every internal node of every elementary tree has exactly two children. The normal form for TAGs is analogous to the Chomsky Normal Form for context-free grammars. It can be shown that an arbitrary TAG G_1 can be converted to an equivalent TAG G_2 in normal form in time proportional to $O(|G_1|)$, where $|G_1|$ = the number of nodes in all elementary trees of $|G_1|$ [VIJA87].

2.2. Sequential recognition of TALs. We now describe the sequential recognition algorithm for TALs given in [VIJA86]. For a TAG G, define a *rule* to be a tuple of the form $\langle conv, node_1, node_2, node_3 \rangle$, where $conv \in \{0, 1, 2, 3\}$. If $conv = 0$ (called a *leaf rule*), then $node_1$ is a leaf node of some elementary tree of G, and $node_2 = node_3 = \lambda$. $LEAF(l)$ denotes the set of leaf rules whose $node_1$ is labeled l. If $conv \in \{1, 2, 3\}$, then $node_1$ is an internal node; such rules are formed by applying the following *convolution* operations ("$-$" denotes a "don't care" value):

$$\langle -, \Gamma, -, - \rangle *_1 \langle -, \Delta, -, - \rangle = \langle 1, \Theta, \Gamma, \Delta \rangle$$

if and only if Δ is adjoinable at Γ, and Θ is identical to Γ except that *constraint* (Θ) = *constraint* (Δ),

$$\langle -, \Gamma, -, - \rangle *_2 \langle -, \Delta, -, - \rangle = \langle 2, \Theta, \Gamma, \Delta \rangle$$

if and only if Θ is the parent of Γ and Δ, Γ is to the left of Δ, and *constraint* (Γ) and *constraint* (Δ) are both type "*SA*,"

$$\langle -\Gamma, -, - \rangle *_3 \langle -, \Delta, -, - \rangle = \langle 3, \Theta, \Gamma, \Delta \rangle$$

if and only if Θ is the parent of Γ and Δ, Γ is to the left of Δ, and *constraint* (Γ) and *constraint* (Δ) are both type "*SA*."

Note that $*_2$ and $*_3$ are actually the same operation except that the *conv* field of the resulting rule has value two or three, respectively. It is convenient to define these two convolutions separately as they simplify the parsing process.

The convolutions can be extended to sets S_1 and S_2 of rules, i.e., $S_1 *_i S_2 = \{R \mid R = R_1 *_i R_2$, for some $R_1 \in S_1$ and some $R_2 \in S_2\}$. We assume that for any set S, $S *_i \varnothing = \varnothing *_i S = \varnothing$. Finally, for any set of rules S, we define $CLOSURE(S)$ to be the value returned by the following function:

function $CLOSURE(S)$;
repeat
    $S_1 \leftarrow S$;
    $S = S \cup [LEAF(\varepsilon) *_2 S] \cup [S *_3 LEAF(\varepsilon)]$;
until $S_1 = S$;
return (S);
end $CLOSURE$;

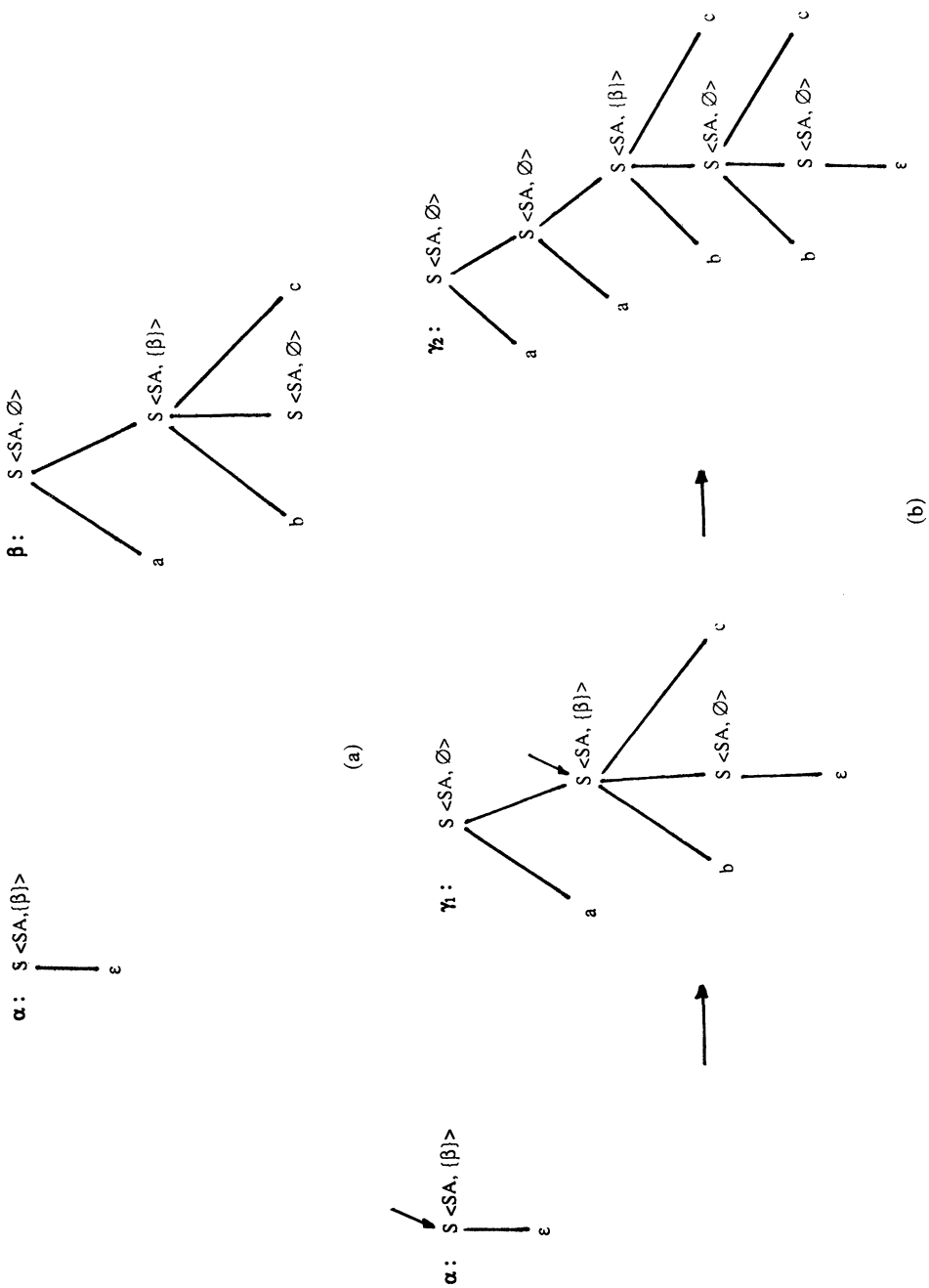


FIG. 2.2. *An example of a* TAG G, *where* $L(G) = \{a^n b^n c^n \mid n \geqq 0\}$.

The function *CLOSURE* is used later to obtain chains of nodes in elementary trees that have children labeled ε. The number of such nodes is bounded above by the size $|G|$ of the grammar; hence, the loop is iterated at most $|G|$ times.

Given a TAG G in normal form and an input string $a_1 a_2 \cdots a_n$, $n \geqq 1$, the Vijayashanker-Joshi dynamic programming algorithm [VIJA86] constructs a four-dimensional recognition matrix A whose elements (or items) are sets of rules. Item $A(i, j, k, l)$, $0 \leqq i \leqq j \leqq k \leqq l \leqq n$, has the property that (see Fig. 2.3):

> $\langle -, \Theta, -, - \rangle \in A(i, j, k, l)$ if and only if Θ is a node in a derived tree γ and the subtree of γ rooted at Θ has a yield given by either $a_{i+1} \cdots a_j Y a_{k+1} \cdots a_l$ (when $j < k$) or $a_{i+1} \cdots a_l$ (when $j = k$).

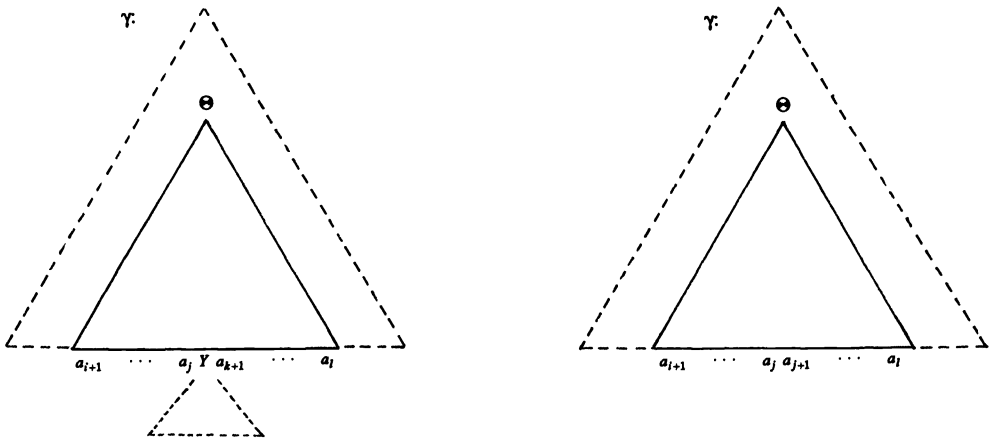


FIG. 2.3. $\langle -, \Theta, -, - \rangle \in A(i, j, k, l)$.

The recognition matrix is computed as follows. Initially, all items are set to the null set $\varnothing$. Then, items $A(i, i, i, i+1)$ and $A(i, i+1, i+1, i+1)$ for $0 \leqq i \leqq n-1$ are set to $CLOSURE\,(LEAF\,(a_{i+1}))$, $A(i, i, i, i)$ for $0 \leqq i \leqq n$ are set to $CLOSURE\,(\cup_{Y \in (N \cup \{\varepsilon\})} LEAF\,(Y))$, and $A(i, i, j, j)$ for $0 \leqq i < j \leqq n$ are set to $CLOSURE\,(\cup_{Y \in N} LEAF\,(Y))$. The rest of the matrix is computed according to the following equation.

(2.1) (1) $$A_1(i, j, k, l) = \bigcup_{i \leqq m \leqq j} \bigcup_{k \leqq p \leqq l} [A(m, j, k, p) *_1 A(i, m, p, l)],$$

(2) $$A_2(i, j, k, l) = \bigcup_{i \leqq m \leqq j} [B(i, m) *_2 A(m, j, k, l)],$$

(3) $$A_3(i, j, k, l) = \bigcup_{k \leqq p \leqq l} [A(i, j, k, p) *_3 B(p, l)],$$

(4) $$A(i, j, k, l) = CLOSURE\,(A_1(i, j, k, l) \cup A_2(i, j, k, l) \cup A_3(i, j, k, l))$$

where $B(0:n, 0:n)$ is an auxiliary matrix such that $B(q, s) = \cup_{q \leqq r \leqq s} A(q, r, r, s)$.

Note that in (2.1), the occurrence of $A(i, j, k, l)$ in the right-hand side of some equation (e.g., $A(m, j, k, p)$ with $m = i$ and $p = l$), respresents the initial value $\varnothing$. We refer to pairs of items occurring in the right-hand side of any of the equations (e.g., $[A(m, j, k, p), A(i, m, p, l)]$) as the *convolving pairs* of $A(i, j, k, l)$.

Sequentially, (2.1) is computed as follows:

for $l = 0$ **to** n **do**
  for $i = l$ **downto** 0 **do** $|{*}SPAN = l - i{*}|$
    for $j = i$ **to** l **do** $|{*}LEFTSPAN = j - i{*}|$
      for $k = l$ **downto** j **do** $|{*}RIGHTSPAN = l - k{*}|$
        compute steps (1)-(4) of (2.1).

The input string $a_1 a_2 \cdots a_n \in L(G)$ if and only if there exists a rule $\langle -, \Theta, -, - \rangle$ in $B(0{:}\ n)$ such that Θ is the root node of some initial tree of G, and *constraint* (Θ) is not of type "*OA*."

The annotated variables *SPAN*, *LEFTSPAN*, and *RIGHTSPAN* are shown in Fig. 2.4. The main point to note is that for a fixed *SPAN*, we progressively decrease the $gap = (SPAN - [LEFTSPAN + RIGHTSPAN])$ dominated by the foot node until it becomes 0, and the nodes under consideration dominate subtrees with terminal yields. One can also verify that items having the same l, *SPAN*, and *gap* do not depend on each other and hence can be computed in any order.

2.3. Sequential parsing of TALs. We now describe a parse (derivation) tree of a string in the tree adjoining language. Suppose that $\langle -, \Theta, -, - \rangle$ is the rule obtained in $B(0, n)$ at the end of the recognition algorithm; i.e., Θ is the root node of some initial tree α of the grammar. Let $\beta_1, \cdots, \beta_6$ be auxiliary trees with root nodes $\Delta_1, \cdots, \Delta_6$, respectively. Consider the following derivation of a certain string from α:

(1) $\Delta_1, \cdots, \Delta_3$ are adjoined at nodes $\Gamma_1, \cdots, \Gamma_3$, respectively, in α.
(2) Δ_4 is adjoined at node Γ_4 in β_1.
(3) Δ_5, Δ_6 are adjoined at nodes Γ_5, Γ_6, respectively, in β_3.

Then, given that *index* $(\Gamma_1) <$ *index* $(\Gamma_2) <$ *index* (Γ_3) in α, and *index* $(\Gamma_5) <$ *index* (Γ_6) in β_3, we can pictorially describe the derivation as a tree of rules resulting from the adjunctions (see Fig. 2.5).

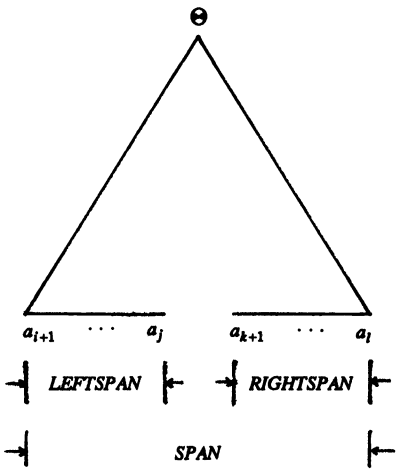


FIG. 2.4. *Dynamic programming parameters for* $A(i, j, k, l)$.

Informally, the root of the parse tree is labeled with the rule $\langle -, \Theta, -, - \rangle$ representing the root node of initial tree α. The children of the root correspond, in left-to-right order, to the adjunctions performed on the initial tree in order of increasing indices of the nodes where the adjunctions take place. For the children of other parse tree

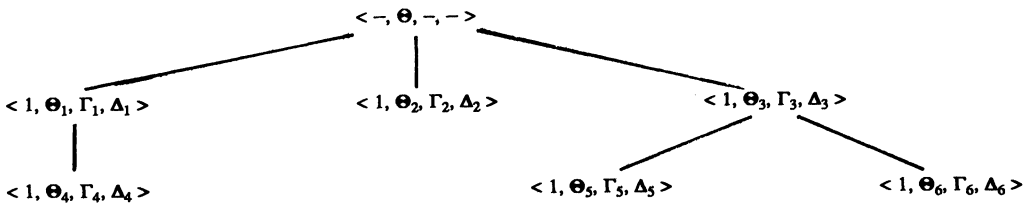


FIG. 2.5. *Parse tree of a* TAG *derivation.*

nodes, the left-to-right order has the same connotation except that adjunctions are now performed on the nodes of the auxiliary tree, whose root is represented by the fourth field (i.e., the Δ_i's) of the rule. A postorder traversal of this parse tree results in a sequence of adjunction rules that we shall call the *bottom-up inside-out parse* (or simply *parse*) of the input string. For the given example, the parse corresponds to

$$\langle 1, \Theta_4, \Gamma_4, \Delta_4\rangle\langle 1, \Theta_1, \Gamma_1, \Delta_1\rangle\langle 1, \Theta_2, \Gamma_2, \Delta_2\rangle\langle 1, \Theta_5, \Gamma_5, \Delta_5\rangle\langle 1, \Theta_6, \Gamma_6, \Delta_6\rangle$$
$$\langle 1, \Theta_3, \Gamma_3, \Delta_3\rangle\langle -, \Theta, -, -\rangle.$$

Note that the structure of the derived initial tree that yields the input string can be fully recovered from the parse described above.

If the input string is valid (i.e., in $L(G)$), a parse of the string can be recovered by searching back through the recognition matrix A and "marking" rules representing adjunctions (i.e., $conv = 1$). The details are given in procedure *SEARCH-FOR-PARSE* below (function *REDUCE* "unravels" the closure of the $A(i, j, k, l)$ computed in step (4) of (2.1)).

Note that only adjunctions ($conv = 1$) are included in *PARSE*. That this list corresponds to a bottom-up inside-out parse is justified by the following proposition.

PROPOSITION 2.1. *Let PARSE contain the sequence* $I_1, I_2, \cdots, I_m$ *at the end of the algorithm. For any* $I_p = (i_p, j_p, k_p, l_p, \langle -, \Theta_p, -, -\rangle)$ *in this sequence, let* $SPAN_p$ *and* $FRONTIER_p$ *denote the values of* $(l_p - i_p)$ *and* $[(j_p - i_p) + (l_p - k_p)]$, *respectively. Then for every pair,* I_p *and* I_q, $1 \leqq p < q \leqq m$, *in the sequence, either one of the following statements hold*:

(1) $l_p < l_q$, or

(2) $l_p = l_q$, $SPAN_p \leqq SPAN_q$ *and* $FRONTIER_p < FRONTIER_q$.

The proposition is proved quite easily by induction on the length of the parse sequence and from the fact that every adjunction creates trees with longer yields (i.e., there are at most n adjunction rules in *PARSE*). This proposition will be used later in the description of the parallel parsing algorithm. Note that if the indices i, j, k, and l were left out of the items of the parse sequence, then we would obtain the bottom-up inside-out representation described earlier.

function *REDUCE* $(i, j, k, l, \langle conv_1, \Theta_1, \Gamma_1, \Delta_1\rangle)$;
if $(conv_1 = 0, 1)$ **then return** $(\langle conv_1, \Theta_1, \Gamma_1, \Delta_1\rangle)$;
$\langle conv, \Theta, \Gamma, \Delta\rangle = \langle conv_1, \Theta_1, \Gamma_1, \Delta_1\rangle$;
done = *false*;
repeat
  if Γ is in *LEAF* (ε) **then**
    find a rule $R = \langle -, \Delta, -, -\rangle$ in $A(i, j, k, l)$
  else
    if Δ is in *LEAF* (ε) **then**
      find a rule $R = \langle -, \Gamma, -, -\rangle$ in $A(i, j, k, l)$
    else *done* = *true*;
  if not (*done*) **then**

```
        ⟨conv, Θ, Γ Δ⟩ = R:
until done;
return (⟨conv, Θ, Γ, Δ⟩);
end REDUCE;

procedure SEARCH-FOR-PARSE (i, j, k, l, ⟨conv1, Θ1, Γ1, Δ1⟩);
⟨conv, Θ, Γ, Δ⟩ = REDUCE (i, j, k, l, ⟨conv1, Θ1, Γ1, Δ1⟩);
case
conv = 0: return;
conv = 1:
    for i ≦ m ≦ j and k ≦ p ≦ l do
        if ⟨−, Γ, −, −⟩ is in A(m, j, k, p) and ⟨−, Δ, −, −⟩ is in A(i, m, p, l) then
            SEARCH-FOR-PARSE (m, j, k, p, ⟨−, Γ, −, −⟩);
            SEARCH-FOR-PARSE (i, m, p, l, ⟨−, Δ, −, −⟩);
            append (i, j, k, l, ⟨conv, Θ, Γ, Δ⟩) to the end of PARSE;
            return;
        endif;
conv = 2:
    for k ≦ p ≦ l and p ≦ m ≦ l do
        if ⟨−, Γ, −, −⟩ is in A(i, j, k, p) and ⟨−, Δ, −, −⟩ is in A(p, m, m, l) ∈ B(p, l) then
            SEARCH-FOR-PARSE (i, j, k, p, ⟨−, Γ, −, −⟩);
            SEARCH-FOR-PARSE (p, m, m, l, ⟨−, Δ, −, −⟩);
        endif; return;
conv = 3:
    for i ≦ m ≦ j and i ≦ p ≦ m do
        if ⟨−, Γ, −, −⟩ is in A(i, p, p, m) ∈ B(i, m) and ⟨−, Δ, −, −⟩ is in A(m, j, k, l) then
            SEARCH-FOR-PARSE (i, p, p, m, ⟨−, Γ, −, −⟩);
            SEARCH-FOR-PARSE (m, j, k, l, ⟨−, Δ, −, −⟩);
            return;
        endif;
endcase;
end SEARCH-FOR-PARSE;

|*MAIN*|
initialize a global variable PARSE to the empty sequence (list);
let ⟨conv, Θ, −, −⟩ be the rule in B(0, n) found in the last step of the recognition
algorithm;
find an item A(0, j, j, n) ∈ B(0, n) containing ⟨conv, Θ, −, −⟩;
call SEARCH-FOR-PARSE (0, j, j, n, ⟨conv, Θ, −, −⟩);
```

3. The processor array model. The parallel machine model is a five-dimensional array of processors numbered $P(0, 0, 0, 0, 0)$ through $P(n, n, n, n, n)$ (where n is the length of the input). Processor $P(a, b, c, d, e)$ is directly connected to other processors via a set of unidirectional *links* $L = \{[\Delta a, \Delta b, \Delta c, \Delta d, \Delta e] | \Delta a, \cdots, \Delta e \in \{-1, 0, 1\}\}$. Link $[\Delta a, \Delta b, \Delta c, \Delta d, \Delta e]$ connects $P(a, b, c, d, e)$ to $P(a+\Delta a, b+\Delta b, c+\Delta c, d+\Delta d, e+\Delta e)$; data through this link can only flow from the former processor to the latter. Moreover, the *delay* along the link is $d = (|\Delta a| + |\Delta b| + |\Delta c| + |\Delta d| + |\Delta e|)$. That is, a data item sent out of the former processor at time t arrives at the latter processor at time $(t+d)$. The assumption that all links in the set L are available to each processor is only made to simplify the presentation of the parallel algorithm. At the expense of additional control logic, L can be reduced to the set of nearest-neighbor links $\{[\pm 1, 0, 0, 0, 0], [0, \pm 1, 0, 0, 0], [0, 0, \pm 1, 0, 0], [0, 0, 0, \pm 1, 0], [0, 0, 0, 0, \pm 1]\}$.

We assume that the processors operate at discrete time steps by means of a global clock. The input to the array is a string of the form $¢a_1a_2 \cdots a_n\$$, where $a_1a_2 \cdots a_n$ represents the string to be parsed. It is fed serially to processor $P(0, 0, 0, 0, 0)$ beginning at time 0; i.e., ¢ is received at time 0, a_i at time i, and \$ at time $(n+1)$.

The parallel algorithm is divided into two distinct phases: *recognition* and *parse recovery*. These phases have the property that in the former, data flows only toward higher-numbered processors, while in the latter, data flows only toward lower-numbered processors. For this reason, it is convenient to describe the computation of the array in terms of forward and reverse *sweeps* that we now define.

Let $d_f(a, b, c, d, e) = (a+b+c+d+e)$ denote the distance of processor $P(a, b, c, d, e)$ from processor $P(0, 0, 0, 0, 0)$. Then, $P(a, b, c, d, e)$ is said to be at *forward sweep s with base-time* t_0 if and only if the processor is currently at timestep $(s+t_0+d_f(a, b, c, d, e))$. Intuitively, the base-time is the timestep at which processor $P(0, 0, 0, 0, 0)$ "signals" the start of a sequence of computation steps C_0, C_1, etc., that are to be performed by all processors of the array. Computation step C_s is performed by every processor at forward sweep s. However, forward sweep s represents different time steps for different processors, as it is defined in terms of the processor's distance from $P(0, 0, 0, 0, 0)$. In particular, if t_1 is the timestep corresponding to forward sweep s of processor P_1, then for a *higher-numbered* adjacent processor P_2, forward sweep s corresponds to timestep $t_2 = t_1 + d$, where d is the delay along the link connecting the two processors. Thus, the result of computation step C_s in P_1 *can* affect computation step C_s in P_2, since the result in the former can be sent to the latter just in time to take part in the latter's computation. Fig. 3.1 illustrates forward sweep s with base-time 0 for the subarray of processors $\{P(a, b, c, d, e) \mid 0 \leqq d, e \leqq b\}$. Observe that a forward sweep is not a "snapshot" of the subarray at a specific timestep since processors are viewed at different times. Moreover, for each processor the next forward sweep $(s+1)$ corresponds to the next timestep.

One advantage of the sweep notion is that certain computations on the array can be described as occurring during *one* particular forward sweep, as opposed to occurring over a sequence of several time steps. As an example, in Fig. 3.1, suppose that at the start of some forward sweep s, every processor $P(a, b, c, d, e)$ holds an item $v(d, e)$.

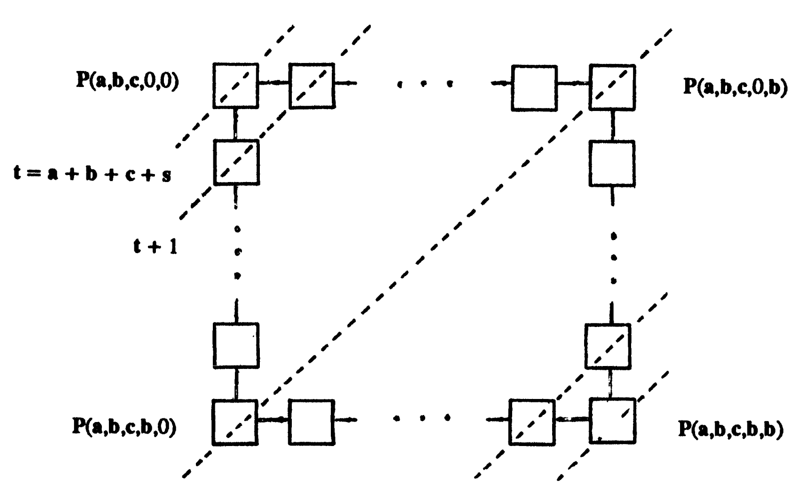


FIG. 3.1. *Forward sweep s with base-time* 0 *for* $\{P(a, b, c, d, e) \mid 0 \leqq d, e \leqq b\}$.

Then, during the same forward sweep, the union $v(d)$ of all items $v(d, e)$ in row d can be computed at processor $P(a, b, c, d, b)$ by sending the items along the horizontal direction. Each processor receives a value from its left-neighboring processor (if it exists), takes the union of this value and the item it holds, and sends the result to the right. The values will be ready at the rightmost processors also at forward sweep s. The operation can be extended to higher dimensions. For example, each processor in Fig. 3.1 can instead receive values from the processors above it and to its left, take the union of these values and the item it holds, and send the result downward and to the right. The union v of all items in the subarray will thus be ready at $P(a, b, c, b, b)$ also at forward sweep s. Because this operation will be used quite frequently in the subsequent sections, we shall refer to it as the *union operation.*

Analogously, a reverse sweep can be defined as follows. Let $d_r(a, b, c, d, e) = 5n - (a+b+c+d+e)$ be the distance of processor $P(a, b, c, d, e)$ from $P(n, n, n, n, n)$. Then, $P(a, b, c, d, e)$ is said to be at *reverse sweep s with base-time* t_0 if and only if the processor is currently at timestep $(s + t_0 + d_r(a, b, c, d, e))$. In other words, a reverse sweep is similar to a forward sweep except that it is initiated from $P(n, n, n, n, n)$. Moreover, lower-numbered processors are viewed at later timesteps than higher-numbered processors.

We end this section with a description of the organization of a processor in the array. Each processor $P(a, b, c, d, e)$ is divided into two processing elements (PEs) denoted $P_0(a, b, c, d, e)$ and $P_1(a, b, c, d, e)$. Each PE has its own small local memory and operates independently of the other. The local memory of each PE consists of a number of *data* and *accumulator* registers as shown in Fig. 3.2. The data registers come in pairs and are partitioned into three register *banks* labeled $R1$, $R2$, and $R3$. The register-pairs within each bank are named as shown in the figure. The left (right) register of a register-pair is referred to by appending the suffix "*.left*" ("*.right*") to the register-pair name, e.g., $R1[0, 0]$*.left* refers to the left register of register-pair $R1[0, 0]$. In addition, there are two accumulator registers named A and B. A few temporary registers are also assumed to be available. The functions of the registers are described in the next section.

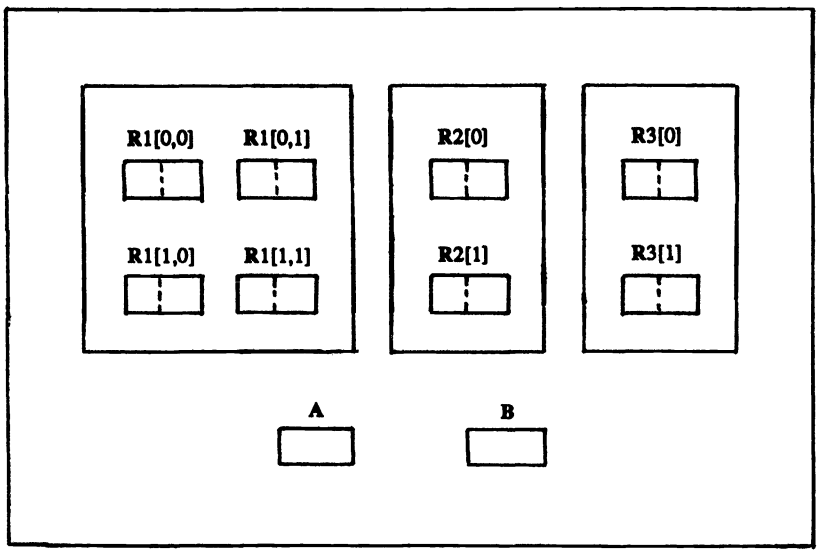


FIG. 3.2. *Local memory of a* PE.

A processor $P(a, b, c, d, e)$ is called *primary* if and only if $b = d = e$. PEs of primary processors are called primary PEs. Primary processors are distinguished in that items of the recognition matrix are computed only in these processors (one in each PE). Note that there are only $O(n^3)$ primary processors.

4. The recognition phase. The *recognition phase* computes the items of the recognition matrix A and determines whether the input is a valid string of the tree-adjoining grammar. It consists of $(n+2)$ consecutive forward sweeps 0 through $(n+1)$, each with base-time $t_0 = 0$. With respect to $P(0, 0, 0, 0, 0)$, these correspond to timesteps 0 to $(n+1)$ during which it reads the input string. The recognition matrix is constructed incrementally; that is, for each new forward sweep only a small new portion of the matrix is computed. Moreover, the computed items are ultimately stored only at the primary processors. As there are only $O(n^3)$ primary processors and $O(n^4)$ items to be computed, every primary processor is assigned $O(n)$ items. The primary processor computes these items at different forward sweeps. Thus, each item is actually mapped onto a specific processor and a specific forward sweep. This *processor-sweep* mapping is described in the next section.

4.1. The processor-sweep mapping. Items $A(i, j, k, l)$, $0 \leqq i \leqq j \leqq k \leqq l \leqq n$, of the recognition matrix are mapped onto processors and sweeps as stated in the following theorem.

THEOREM 4.1. *$A(i, j, k, l)$ is computed and stored in register A of primary* PE *$P_x(a, b, c, b, b)$ at forward sweep s, where*

$$s = l, \quad a = (l-i), \quad b = (j-i)+(l-k),$$

$$[c, x] = \begin{cases} [(l-k), 0] & \text{if } (j-i) \leqq (l-k), \\ [(j-i), 1] & \text{if } (j-i) \geqq (l-k). \end{cases}$$

We refer $P_x(a, b, c, b, b)$ as the *target primary* PE (or simply *target*) of $A(i, j, k, l)$. Every $A(i, j, k, l)$ has exactly one target, except when $(j-i) = (l-k)$ in which case it has as targets both $P_x(a, b, c, b, b)$, $x \in \{0, 1\}$. It is easy to verify that in the latter case, $2c = b$. Several items may have the same target; however, they are computed by the target at different forward sweeps. For example, $A(0, 1, 1, 3)$, $A(1, 2, 2, 4)$, and $A(2, 3, 3, 5)$ have the same target $P_0(3, 3, 2, 3, 3)$, but are computed at forward sweeps 3, 4, and 5, respectively.

Theorem 4.1 implies that at forward sweeps s, only PEs $P_x(a, b, c, b, b)$ for which $0 \leqq b \leqq a \leqq s$ and $\lceil b/2 \rceil \leqq c \leqq b$ are active (i.e., compute items). Moreover, they compute only items $A(i, j, k, l)$ for which $l = s$. Figure 4.1 illustrates the active primary PEs and the items they compute for forward sweeps 0 through 4.

Items $B(q, s)$, $0 \leqq q \leqq s \leqq n$, of the auxiliary matrix are also mapped onto PEs and sweeps as stated below.

THEOREM 4.2. *$B(q, s)$ is computed and stored in register B of primary* PE*s $P_x(a, a, a, a, a)$, $x \in \{0, 1\}$, at forward sweep s, where $a = (s-q)$.*

$B(q, s)$ is easy to compute. To see this, suppose inductively that Theorem 4.1 holds for items in the set $[A(q, r, r, s) \mid q \leqq r \leqq s]$. Thus, their values are available at forward sweep s in the contiguous block of primary processors $\{P(a, a, c, a, a) \mid a = (s-q)$ and $\lceil a/2 \rceil \leqq c \leqq a\}$. By performing a union operation over these values at forward sweep s, their union $B(q, s)$ can be computed and stored in both PEs of processor $P(a, a, a, a, a)$ during the same sweep.

Computing the $A(i, j, k, l)$'s is also simple. Suppose that $P_x(a, b, c, b, b)$ is the target of item $A(i, j, k, l)$. In order to compute this item, the algorithm makes sure that

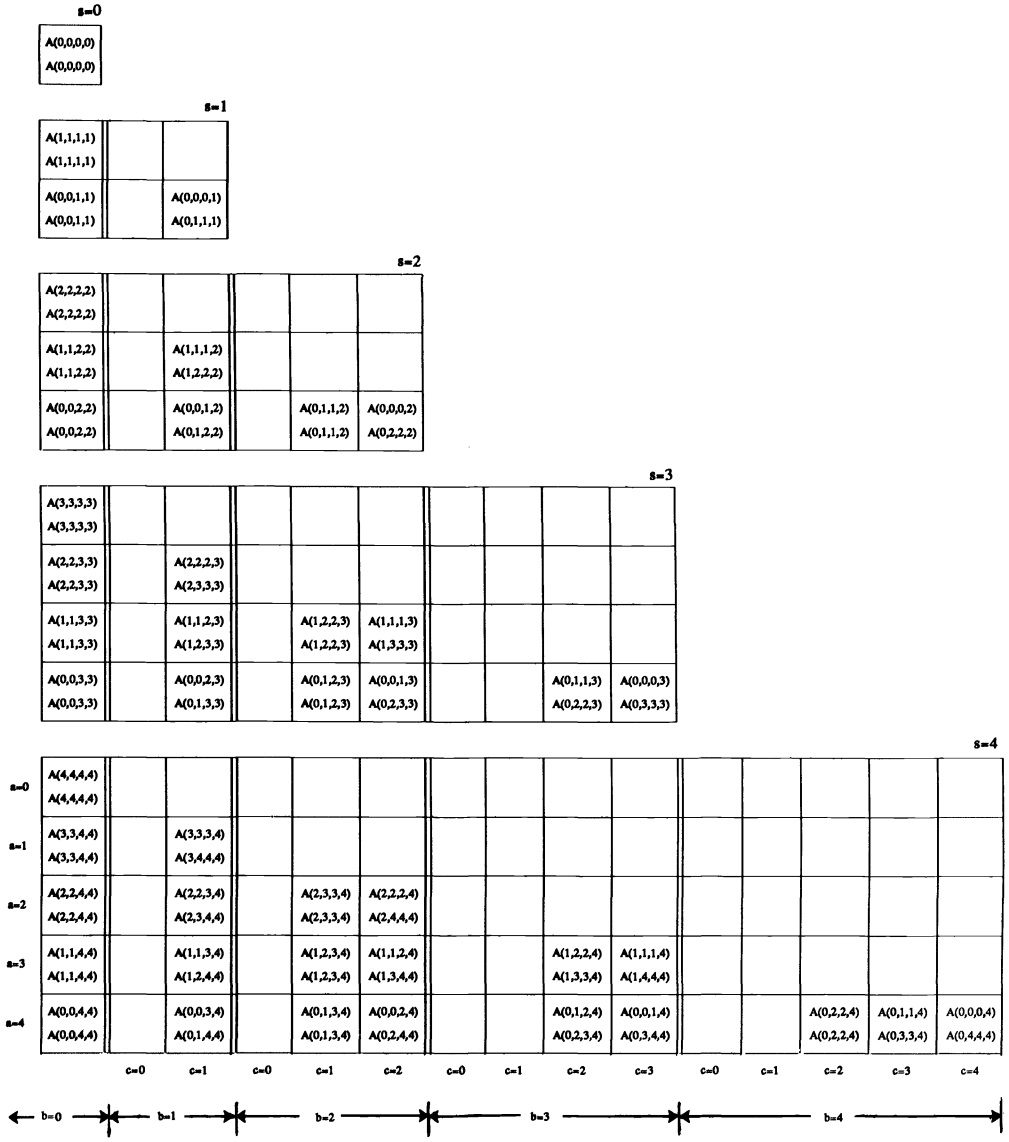


FIG. 4.1. *Items computed by primary processors for forward sweeps* 0 *to* 4.

at forward sweep $s = l$, all convolving pairs of $A(i, j, k, l)$ are already available in the subarray of PEs $P_x(a, b, c, *, *) = \{P_x(a, b, c, d, e) \mid d, e \leqq b\}$. During the same sweep, the required convolution operations are performed by each PE on its local convolving pairs, then the union of the partial results is obtained (via a union operation) and stored in primary PE $P_x(a, b, c, b, b)$. The resulting value is $S = A_1(i, j, k, l) \cup A_2(i, j, k, l) \cup A_3(i, j, k, l)$ (see equation (2.1)). $A(i, j, k, l)$ is then simply $CLOSURE(S)$.

The tricky part is how to "distribute" the convolving pairs of $A(i, j, k, l)$ among the PEs of subarray $P_x(a, b, c, *, *)$. We first give an informal description. The convolving pairs of $A(i, j, k, l)$ can be naturally subdivided into the convolving pairs of its *partial items* $A_r(i, j, k, l)$, $1 \leqq r \leqq 3$.

First consider $A_1(i, j, k, l) = \cup_{i \leq m \leq j} \cup_{k \leq p \leq l} A(m, j, k, p) *_1 A(i, m, p, l)$. Construct a matrix $Q(i:j, k:l)$, such that $Q(m, p)$ contains the pair of items $[A(m, j, k, p), A(i, m, p, l)]$. Now "fold" Q along the center row (or between the two center rows if the number of rows is even). Fold Q once more, this time along the center column. Figure 4.2 gives an illustration for the case $A(i, j, k, l) = A(1, 4, 5, 7)$. The folding results in a $\lceil (j-i)/2 \rceil + \lceil (l-k)/2 \rceil$ matrix Q', as shown in Fig. 4.2. Now, index the rows and columns of Q' from 0, 1, 2, etc. Then the elements of this matrix are mapped onto the PEs of subarray $P_x(a, b, c, *, *)$ such that every pair in $Q'(v, w)$ appears in PE $P_x(a, b, c, b-v, b-w)$. For example, since the primary PE of $A(1, 4, 5, 7)$ is $P_1(6, 5, 3, 5, 5)$, (then the convolving pairs in $Q'(1, 0)$ would appear in $P_1(6, 5, 3, 4, 5)$.

There is a natural correspondence between the convolving pairs mapped onto a specific PE and the register-pairs $R1[y, z]$ of this PE. Each convolving pair can be identified with the "quadrant" yz, $y, z \in \{0, 1\}$, it belonged to in the original unfolded matrix Q, where the quadrants are those induced by the lines where the folding took place (see Fig. 4.2). For example, pairs $[A(2, 4, 5, 5), A(1, 2, 5, 7)]$, $[A(2, 4, 5, 7), A(1, 2, 7, 7)]$, $[A(3, 4, 5, 5), A(1, 3, 5, 7)]$, and $[A(3, 4, 5, 7), A(1, 3, 7, 7)]$

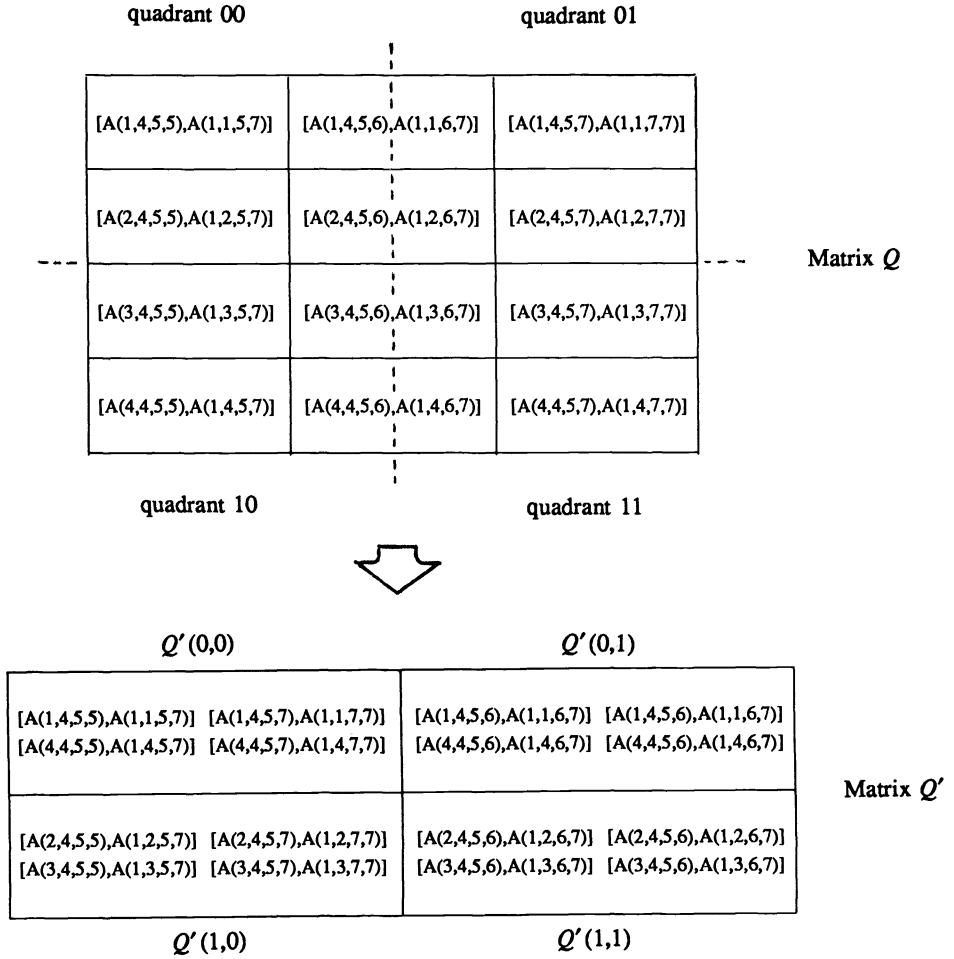


FIG. 4.2. *Mapping the convolving pairs of* $A_1(i, j, k, l) = A_1(1, 4, 5, 7)$.

of $Q'(1, 0)$ are in quadrants 00, 01, 10, and 11, respectively. (By convention, if a pair lies on the boundary of two or more quadrants, it is assumed to be in all such quadrants; e.g., $[A(1, 4, 5, 6), A(1, 1, 6, 7)]$ is in quadrants 00 and 01.) The rule for mapping convolving pairs onto register-pairs is as follows: a pair in quadrant yz is mapped onto register-pair $R1[y, z]$ (with obvious interpretation that the left and right terms of the convolving pair goes into the left and right registers, respectively, of the register-pair). For example, in Fig. 4.2, the convolving pairs in $Q'(1, 0)$ are mapped onto PE $P_1(6, 5, 3, 4, 5)$ such that $[A(2, 4, 5, 5), A(1, 2, 5, 7)]$ is stored in register-pair $R1[0, 0]$, $[A(2, 4, 5, 7), A(1, 2, 7, 7)]$ in register-pair $R1[0, 1]$, etc.

The mapping for the convolving pairs of partial item $A_2(i, j, k, l) = \cup_{i \leqq m \leqq j} B(i, m) *_2 A(m, j, k, l)$ is illustrated in Fig. 4.3. A linear array $R(i:j)$ is constructed such that $R(m)$ contains the pair $[B(i, m), A(m, j, k, l)]$, which is then folded along the center column to yield a new array R'. If the rows of R' are indexed 0, 1, 2, etc., then pairs occurring in $R'(\nu)$ are mapped onto PE $P_x(a, b, c, b-\nu, b)$ such that the pair coming from half $y \in \{0, 1\}$ of R is assigned to register-pair $R2[y]$ of this PE.

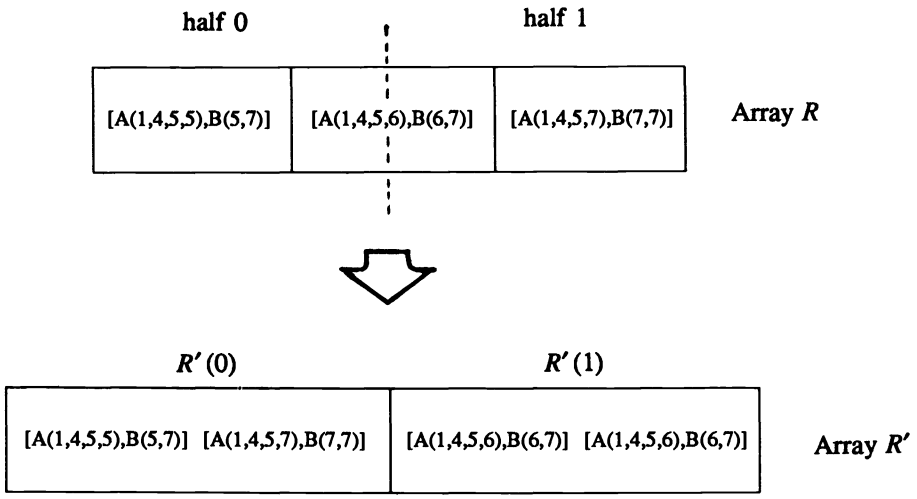


FIG. 4.3. *Mapping the convolving pairs of* $A_2(i, j, k, l) = A_2(1, 4, 5, 7)$.

The mapping for the convolving pairs of the third partial item $A_3(i, j, k, l) = \cup_{k \leqq p \leqq l} A(i, j, k, p) *_3 B(p, l)$ is similar (see Fig. 4.4). The linear array $S(p:l)$ such that $S(p)$ contains $[A(i, j, k, p), B(p, l)]$ is folded to yield a new array S'. Then, convolving pairs occurring in $S'(w)$ are mapped onto PE $P_x(a, b, c, b, b-w)$, with the pair coming from half $z \in \{0, 1\}$ of S assigned to register-pair $R3[z]$ of the PE.

The mapping described above is formalized in the following lemma.

LEMMA 4.1. *At forward sweep* s, *the convolving pairs of* $A(i, j, k, l)$ *are stored in subarray* $P_x(a, b, c, *, *)$ *such that*

(1) $[A(m, j, k, p), A(i, m, p, l)]$ *is in register-pair* $R1[y, z]$ *of* PE $P_x(a, b, c, d, e)$,

(2) $[B(i, m), A(m, j, k, l)]$ *is in register-pair* $R2[y]$ *of* PE $P_x(a, b, c, d, b)$,

(3) $[A(i, j, k, p), B(p, l)]$ *is in register-pair* $R3[z]$ *of* PE $P_x(a, b, c, d, e)$,

where

$$s = l, \quad a = (l-i), \quad b = (j-i)+(l-k),$$

$$[c, x] = \begin{cases} [(l-k), 0] & \text{if } (j-i) \leqq (l-k), \\ [(j-i), 1] & \text{if } (j-i) \geqq (l-k), \end{cases}$$

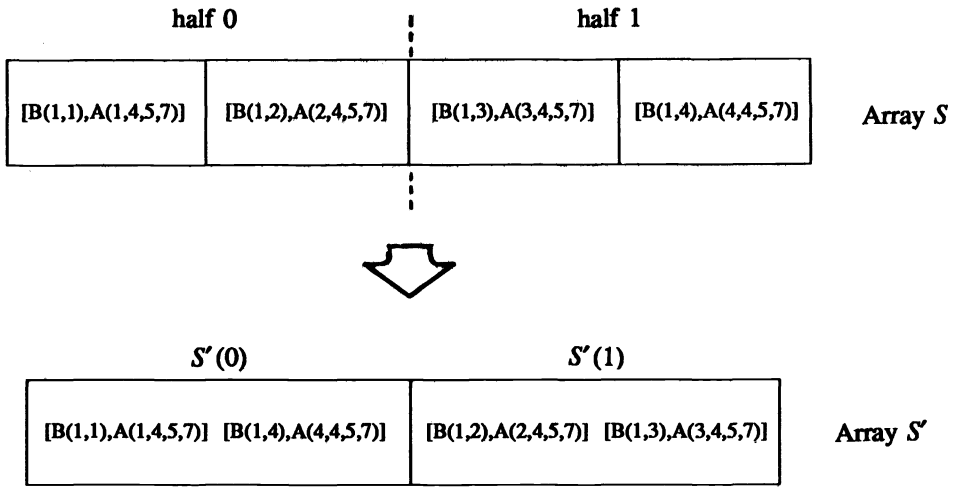


FIG. 4.4. *Mapping the convolving pairs of* $A_3(i, j, k, l) = A_3(1, 4, 5, 7)$.

$$[d, y] = \begin{cases} [(j-m)+(l-k), 0] & \text{if } 0 \leqq 2m \leqq (i+j), \\ [(m-i)+(l-k), 1] & \text{if } 2j \geqq 2m \geqq (i+j), \end{cases}$$

$$[e, z] = \begin{cases} [(j-i)+(l-p), 0] & \text{if } 0 \leqq 2p \leqq (k+l), \\ [(j-i)+(p-k), 1] & \text{if } 2l \geqq 2p \geqq (k+l). \end{cases}$$

Observe that because of the folding, some data registers of a PE may store the same item. It can be shown that this happens only for certain "special" PEs. In particular, we have the following fact.

FACT 4.1. Let $P_x(a, b, c, d, e)$ be a PE. Then at any forward sweep,

(1) If ($x = 0$ and $2d = b + c$) or ($x = 1$ and $2d = 2b - c$), $R1[0, z].t = R1[1, z].t$ and $R2[0].t = R2[1].t$, where $t \in \{left, right\}$;

(2) If ($x = 0$ and $2e = 2b - c$) or ($x = 1$ and $2e = b + c$), $R1[y, 0].t = R1[y, 1].t$ and $R3[0].t = R3[1].t$, where $t \in \{left, right\}$;

(3) *If* $2c = b$, every data register r of $P_x(a, b, c, d, e)$ has the same contents as data register r of $P_{\bar{x}}(a, b, c, d, e)$.

Thus, within each PE, the data registers form equivalence classes depending on the PE's indices. Moreover, if $2c = b$, every data register of the PE is equivalent to the corresponding data register of the other PE sharing the same indices. (Note that a PE may satisfy more than one of the relations listed above.) For a data register r of PE P, we denote by $eq(P, r)$ (or simply $eq(r)$ if P is understood) the set of data registers equivalent to r.

4.2. The routing scheme. We now describe how items are routed through the processor array such that the data registers of each PE are properly updated according to Lemma 4.1.

The routing scheme is embodied in the *routing table* shown in Table 4.1. A copy of Table 4.1 is stored in every processor of the array. Each data register has an associated set of links that defines the directions along which items stored in this register are received and forwarded. Each link $[\Delta a, \Delta b, \Delta c, \Delta d, \Delta e]$ represents both an *input* link $[-\Delta a, -\Delta b, -\Delta c, -\Delta d, -\Delta e]$ from which the item is received, and an *output* link $[+\Delta a, +\Delta b, +\Delta c, +\Delta d, +\Delta e]$ through which the item is forwarded. Observe that the flow of data is only from lower-numbered to higher-numbered processors. This is a consequence of the "folding" technique used in Lemma 4.1.

TABLE 4.1
The routing table.

Register	PE_0 Links	PE_1 Links
$R1[0,0] \cdot left$	$[1,1,1,1,1]^+$	$[1,1,0,1,1]^+$
	$[1,1,0,0,1]$	$[1,1,1,0,1]$
$R1[0,1] \cdot left$	$[1,1,1,1,0]^+$	$[1,1,0,1,0]^+$
	$[1,1,0,0,1]$	$[1,1,1,0,1]$
$R1[1,0] \cdot left$	$[1,1,1,1,1]^+$	$[1,1,0,1,1]^+$
	$[1,1,0,1,1]$	$[1,1,1,1,1]$
$R1[1,1] \cdot left$	$[1,1,1,1,0]^+$	$[1,1,0,1,0]^+$
	$[1,1,0,1,1]$	$[1,1,1,1,1]$
$R1[0,0] \cdot right$	$[0,1,1,1,0]$	$[0,1,0,1,0]$
	$[0,1,0,1,1]$	$[0,1,1,1,1]$
$R1[0,1] \cdot right$	$[0,1,1,1,1]$	$[0,1,0,1,1]$
	$[0,1,0,1,1]$	$[0,1,1,1,1]$
$R1[1,0] \cdot right$	$[0,1,1,1,0]$	$[0,1,0,1,0]$
	$[0,1,0,0,1]$	$[0,1,1,0,1]$
$R1[1,1] \cdot right$	$[0,1,1,1,1]$	$[0,1,0,1,1]$
	$[0,1,0,0,1]$	$[0,1,1,0,1]$
$R2[0] \cdot left$	$[1,1,1,1,1]^+$	$[1,1,0,1,1]^+$
	$[1,0,0,0,0]^+$	$[1,0,0,0,0]^+$
	$[0,1,0,1,1]$	$[0,1,1,1,1]$
$R2[1] \cdot left$	$[1,1,1,1,1]^+$	$[1,1,0,1,1]^+$
	$[1,0,0,0,0]^+$	$[1,0,0,0,0]^+$
	$[0,1,0,0,1]$	$[0,1,1,0,1]$
$R2[0] \cdot right$	$[1,1,0,0,1]$	$[1,1,1,0,1]$
$R2[1] \cdot right$	$[1,1,0,1,1]$	$[1,1,1,1,1]$
$R3[0] \cdot left$	$[1,1,1,1,1]^+$	$[1,1,0,1,1]^+$
$R3[1] \cdot left$	$[1,1,1,1,0]^+$	$[1,1,0,1,0]^+$
$R3[0] \cdot right$	$[1,1,0,1,1]$	$[1,1,1,1,1]$
	$[1,0,0,0,0]$	$[1,0,0,0,0]$
	$[0,1,1,1,0]$	$[0,1,0,1,0]$
$R3[1] \cdot right$	$[1,1,0,1,1]$	$[1,1,1,1,1]$
	$[1,0,0,0,0]$	$[1,0,0,0,0]$
	$[0,1,1,1,1]$	$[0,1,0,1,1]$

Each item is routed as a tuple of the form $\langle v, r, x\rangle$, where v is the value of the item, r is the name of the register from which it originates, and x ($=0$ to 1) indicates the PE within the processor where register r is located. Procedures *UPDATE-REGISTERS* and *ROUTE-REGISTERS* below describe how the updating and routing of data registers are carried out.

UPDATE-REGISTERS. To update a data register, say $R1[0,0].left$ of PE $P_0(a, b, c, d, e)$, the PE first checks whether from any of input links $[-1,-1,-1,-1,-1]$ and $[-1,-1,0,0,-1]$ associated with this register, there is a tuple of the form $\langle v, R1[0,0].left, 0\rangle$. If such a tuple exists, the PE puts the value v in all registers in the same equivalence class as register $R1[0,0].left$. A PE may receive more than one tuple targeted for the same register; however, the values associated with the tuples will always be the same.

ROUTE-REGISTERS. To route the value v stored in a data register, again say $R1[0,0].left$ of $P_0(a, b, c, d, e)$, the PE first creates the tuple $\langle v, R1[0,0].left, 0\rangle$, then sends it out via the associated output links $[+1,+1,+1,+1,+1]$ and $[+1,+1,0,0,+1]$.

If in the routing table, an output link is labeled "+", the PE "waits" one clock cycle before sending out the tuple, e.g., by first storing it into a temporary register. (Intuitively, tuples sent out via links labeled by "+" reach the destination PE one forward sweep later.)

The routing table guarantees that items reach the right data registers at the right times, as specified by Lemma 4.1. The interested reader is referred to the Appendix for the derivation of the routing table.

For the routing scheme described above to work properly, procedures *UPDATE-REGISTERS* and *ROUTE-REGISTERS* should only be executed by *active* PEs (i.e., PEs that store items as specified by Lemma 4.1). The following fact is easy to verify.

FACT 4.2. PE $P_x(a, b, c, d, e)$ is active at forward sweep s if and only if the following conditions hold:

(1) $b \leqq a \leqq s$;
(2) $b \leqq 2c \leqq 2b$;
(3) If $x = 0$ then $b + c \leqq 2d \leqq 2b$ and $2b - c \leqq 2e \leqq 2b$;
(4) If $x = 1$ then $2b - c \leqq 2d \leqq 2b$ and $b + c \leqq 2e \leqq 2b$.

Inactive PEs do not participate in any computation or routing of items. The routing table may in fact forward items from active to inactive PEs (we found this necessary to make the routing table uniform for all processors). However, by definition, inactive PEs receiving items simply discard them.

4.3. The algorithm. Procedure *RECOGNIZE* below specifies the steps performed by every active PE at each forward sweep. Each call to the procedure represents one clock cycle.

procedure *RECOGNIZE* ($P_x(a, b, c, d, e)$).
if (*active*) **then**
case
|*Compute boundary items.*|
$P_x(0, 0, 0, 0, 0)$: receive input symbol "a" and send it to $P_x(1, 1, 1, 1, 1)$ with a sweep delay; $A \leftarrow CLOSURE\left(\bigcup_{Y \in (N \cup \{\varepsilon\})} LEAF(Y)\right)$;

$P_x(a, 0, 0, 0, 0), a > 0$: $A \leftarrow CLOSURE\left(\bigcup_{Y \in N} LEAF(Y)\right)$;

$P_x(1, 1, 1, 1, 1)$: receive input symbol "a" from $P_x(0, 0, 0, 0, 0)$,
$A \leftarrow CLOSURE\ (LEAF\ (a))$;
otherwise:
|*Compute other items.*|
UPDATE-REGISTERS:
|*Compute partial results and perform union operation.*|
|*Ar, $1 \leqq r \leqq 3$, are temporaries.*|

$A1 \leftarrow \bigcup_{y,z \in \{0,1\}} R1[y, z].left *_1 R1[y, z].right$;

$A2 \leftarrow \bigcup_{y \in \{0,1\}} R2[y].left *_2 R2[y].right$;

$A3 \leftarrow \bigcup_{z \in \{0,1\}} R3[z].left *_3 R3[z].right$;

```
            A ← ⋃ (1≤r≤3) Ar ∪ [A of P_x(a, b, c, d−1, e)] ∪ [A of P_x(a, b, c, d, e−1)];
            if a primary PE then A ← CLOSURE (A);
        endcase;
        if not a primary PE then
            send A to P_x(a, b, c, d+1, e) and P_x(a, b, c, d, e+1)
        else
            |*Prepare newly computed A-item for routing.*|
            eq(R1[0, 1].left) ← A;
            eq(R1[1, 0].right) ← A;                                   (1)
            eq(R2[0].right) ← A;
            eq(R3[1].left) ← A;
            |*Compute B-item and prepare for routing.*|
            if P_x(a, a, c, a, a) then
                B ← A ∪ [B of P_x(a, a, c−1, a, a)];
                send B to P_x(a, a, c+1, a, a);
            endif;
            if P_x(a, a, a, a, a) then
                B ← B ∪ [B of P_x̄(a, a, a, a, a)];
                if x = 0 then
                    eq(R3[0].right) ← B                               (2)
                else
                    eq(R2[1].left) ← B;
            endif;
        endif;
        ROUTE-REGISTERS;
        clear all data registers and accumulators;
    endif;
end RECOGNIZE;
```

The assignment statements labeled (1) and (2) update the data registers that are supposed to hold the A or B item, respectively, computed at the PE (initially, all such registers would contain $\varnothing$). This is done so that these newly computed items can be forwarded to other PEs by procedure *ROUTE-REGISTERS*.

At forward sweep $(n+1)$ processor $P(0, 0, 0, 0, 0)$ sends a "completion" signal to all other processors. When the signal reaches $P(n, n, n, n, n)$, this processor checks whether in $B(0, n)$ (stored in the B register of either of its PEs) there is a rule $\langle -, \Theta, -, - \rangle$ such that Θ is the root node of some initial tree of the grammar and *constraint* (Θ) is not of type "*OA*." If there is such a rule, it initiates the parse recovery phase described in the next section; otherwise, it sends back a "reject" signal to $P(0, 0, 0, 0, 0)$ and the computation halts.

If the input string is valid, the recognition phase ends in processor $P(n, n, n, n, n)$ at forward sweep $(n+1)$. In terms of clock cycles, this corresponds to timestep $(6n+1)$, which is linear in the length of the input string.

4.4. A finite-state implementation. It is clear that only a finite amount of information is stored in the local memory of each PE since each register stores sets whose size depends only on the underlying grammar and not on the length of the input string.

Similarly, processor indices need not be stored as there are only a finite number of different processor (or PE) "types" that need to be distinguished. These include the processors specifically referred to in procedure *RECOGNIZE* (e.g., $P(a, 0, 0, 0, 0)$,

$P(1,1,1,1,1)$, $P(a,a,c,a,a)$, primary processors, etc.), as well as those whose indices satisfy the relations given in Fact 4.1. These processors can be "marked" during the zeroth forward sweep by propagating appropriate "control signals" through the array. For instance, to mark all primary processors (i.e., $P(a,b,c,d,e)$ such that $b=d=e$), the following steps can be performed. When $P(0,0,0,0,0)$ receives input symbol $\not\in$, it marks itself as *primary*. It then sends a signal to all processors $\{P(a,0,0,0,0)\}$ via the $[1,0,0,0,0]$ links, and in turn each $P(a,0,0,0,0)$ sends the signal to processors $\{P(a,b,0,b,b)\}$ via the $[0,1,0,1,1]$ links. Finally, each $P(a,b,0,b,b)$ sends the signal to processors $\{P(a,b,c,b,b)\}$ via the $[0,0,1,0,0]$ links. All processors receiving the signal mark themselves as *primary*. Similar schemes can be used to mark other processor types.

Activating PEs at the right times (see Fact 4.2) can be done in a similar way. Observe that a PE $P_x(a,b,c,d,e)$ satisfying conditions (2)-(4) of fact 4.2 first becomes active at forward sweep a, and remains active till the end of the recognition phase. Thus, it is only necessary to mark such processors at the first sweep they become active. Informally, this can be accomplished as follows. During the zeroth forward sweep, mark all PEs (say by "*") satisfying conditions (2)-(4) of Fact 4.2 using a scheme similar to the one described in the previous paragraph. In addition, perform the following steps. When $P(0,0,0,0,0)$ receives $\not\in$ at forward sweep zero, mark this processor by "#." From $P(0,0,0,0,0)$, send "#" to all processors $\{P(a,0,0,0,0)\}$ via the $[1,0,0,0,0]^+$ links (the "+" indicating a sweep delay). Thus, processor $P(a,0,0,0,0)$ receives "#" at forward sweep a. Finally, during the same sweep when $P(a,0,0,0,0)$ receives "#," send "#" to all other processors in the subarray $P(a,*,*,*,*)$. PE's marked both "*" and "#" are labeled *active*.

Processor $P(n,n,n,n,n)$ can also be determined without knowing n. This is because $P(n,n,n,n,n)$ is the only processor of the form $P(a,a,a,a,a)$ (which is specially marked) that becomes active and receives the "completion" signal (discussed in the previous subsection) in consecutive forward sweeps.

Finally, the neighboring processors of $P(a,b,c,d,e)$ can be reduced to those in the set $[P(a\pm1,0,0,0,0), P(0,b\pm1,0,0,0), \cdots, P(0,0,0,0,e\pm1)\}$ as follows: to send data via link $[\Delta a, \Delta b, \Delta c, \Delta d, \Delta e]$, the data is instead sent via the sequence of links $[\Delta a,0,0,0,0], [0,\Delta b,0,0,0], \cdots, [0,0,0,0,\Delta e]$. The delay is the same in either case, namely, $(\Delta a+\Delta b+\Delta c+\Delta d+\Delta e)$.

5. The parse recovery phase. A parse of a valid input string can be recovered in linear time by executing the *parse recovery phase*. This phase is initiated upon completion of the recognition phase, i.e., once PE $P(n,n,n,n,n)$ has determined that the input string is valid. Recall that this happens at forward sweep $(n+1)$, or equivalently, at timestep $(6n+1)$. Starting at this timestep, PE $P(n,n,n,n,n)$ starts a sequence of $(n+1)$ *reverse* sweeps 0 through n, during which adjunction rules that make up a parse of the input string are "marked."

The parse recovery phase consists of three concurrent subphases: *regeneration*, *marking*, and *outputting*. Each subphase begins at reverse sweep 0 and ends at reverse sweep n.

The marking subphase is a parallelization of procedure *SEARCH-FOR-PARSE* described in § 2; i.e., it searches back through the recognition matrix A and "marks" the parse rules. However, since the recognition phase builds A incrementally, only a small portion of the the matrix (in particular, $\{A(i,j,k,l) \mid l=n\}$) would actually be present in the primary processors of the array at the start of the parse recovery. Thus, the "lost" items should somehow be recovered to proceed with the marking process.

This can be done in a simple way: the idea is to "regenerate" the configurations of the array (i.e., contents of the processor registers) at forward sweeps $0, 1, \cdots, n$ in *reverse* order. That is, given the array configuration at the end of forward sweep n, we regenerate, in one reverse sweep, its configuration at the end of forward sweep $n-1$, then from this its configuration at the end of forward sweep $n-2$, etc. The regeneration subphase accomplishes this task.

5.1. The regeneration subphase. The regeneration subphase is carried out during reverse sweeps 0 through n. During reverse sweep s, $0 \leqq s \leqq n$, the array "reconfigures" itself so that the values of the data registers of every processor are the same as those at the end of forward sweep $(n-s)$. (Note that forward sweep $n+1$ does not affect the data registers of the processors.) This is accomplished by routing the items stored in the registers in the reverse of the directions they took during the recognition phase (i.e., the data flows from higher-numbered to lower-numbered processors).

We now examined in detail how to make the above scheme work. Consider the sets of data registers $S_1^l = [R1[y, z].left\}$, $S_2^l = \{R2[y].left\}$, and $S_3^l = \{R3[z].left\}$ of some PE P. From Table 4.1 observe that a register in any of these sets is routed to an output link with superscript "+". The extra delay associated with this link implies that if the value of the register is routed at forward sweep s, then it reaches the destination PE at forward sweep $(s+1)$. That is, we have Fact 5.1.

FACT 5.1. Let v be the contents of an S_i^l-register, $i = 1, 2, 3$, of PE P at forward sweep s. Then, at forward sweep $(s+1)$, there is a copy of v in some S_i^l-register of a PE adjacent to P.

Fact 5.1 implies that given the value of the S_i^l-registers at forward sweep $(s+1)$, their values at forward sweep s can be regenerated by simply reversing the directions of the routes specified by the routing scheme.

Consider now the sets $S_1^r = \{R1[y, z].right\}$, $S_2^r = \{R2[y].right\}$, and $S_3^r = \{R3[z].right\}$. From the routing Table 4.1, it is clear that because of the absence of the superscript "+" in the output links for any register in these sets, the value of the register at forward sweep s reaches the destination PE also at forward sweep s. In other words, the register's value is effectively lost in the next forward sweep. Thus, it should somehow be stored before the next forward sweep is executed. We make the following observations.

FACT 5.2. For every PE $P = P_x(a, b, c, d, e)$:

(1) The contents of an S_1^r-register of P at forward sweep s has copy in some S_1^r-register of a PE in subarray $P(a, a, *, *, *)$ at forward sweep s.

(2) The contents of an S_2^r-register of P at forward sweep s has a copy in some S_2^r-register of a PE in subarray $P(s, *, *, *, *)$ at forward sweep s.

(3) The contents of an S_3^r-register of P at forward sweep s has a copy in some S_3^r-register of a *PE* in subarray $P(s, s, *, *, *)$ at forward sweep s.

Fact 5.2 reveals how the register values should be stored. In particular, Facts 5.2(2) and (3) require us to store only the contents of the S_2^r-registers of PEs in subarray $P(s, *, *, *, *)$ and the S_3^r-registers of PEs in subarray $P(s, s, *, *, *)$ at the end of forward sweep s. This is enough to guarantee that the contents of the S_2^r- and S_3^r-registers of all other PEs active at forward sweep s will be remembered since they have copies in these PEs. Observe that, during the recognition phase, PEs in these subarrays first become active at forward sweep s so that the PEs in effect "know" when to store the register values. More precisely, each PE of the form $P_x(s, b, c, d, e)(P_x(s, s, c, d, e))$ has *duplicate* S_2^r (S_3^r)-registers (see Fig. 5.1). When the PE first becomes active, it copies the contents of its S_2^r (S_3^r)-registers into the corresponding duplicate registers. The

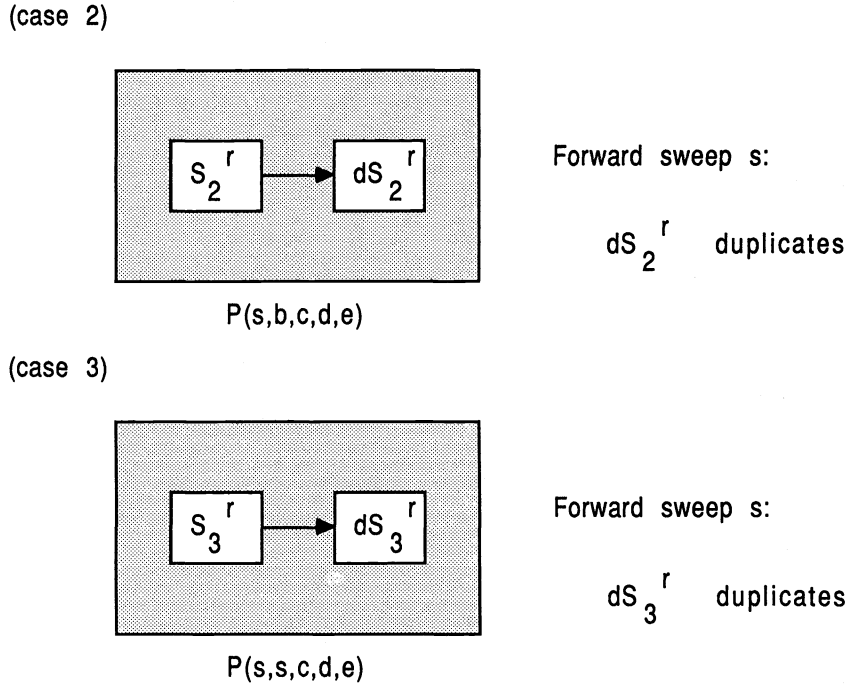


FIG. 5.1. *At forward sweep s, processors* $P(s, b, c, d, e)(P(s, s, c, d, e))$ *copy updated contents of register* $S_2^r(S_3^r)$ *into duplicate registers* $dS_2^r(dS_3^r)$.

contents of the duplicate registers then remain unchanged until the time when forward sweep s needs to be regenerated during parse recovery (which happens at reverse sweep $n-s$). At that time, the original registers can be restored from the duplicates and their values routed in the reverse of the directions specified by the routing scheme.

The situation described by Fact 5.2(1) is somewhat more complicated in that every PE of the form $P_x(a, a, c, d, e)$ must store the contents of its S_1^r-registers at forward sweeps a, $a+1, \cdots, n$ starting from the sweep at which it first became active. Thus, the above technique would not work.

For each PE $P_x(a, a, c, d, e)$, define the sequence of PEs $P_x(a, a, c, d, e)$, $P_x(a+1, a, c, d, e), \cdots, P_x(n, a, c, d, e)$ as the *chain* at $P_x(a, a, c, d, e)$. Clearly, there are $n-a+1$ PEs in the chain; this number coincides with the number of forward sweeps during which $P_x(a, a, c, d, e)$ is active. Moreover, the only PE in the chain for which the first two indices are the same is $P_x(a, a, c, d, e)$. Hence, it is the only PE in the chain that needs to store its S_1^r-registers. It should be evident that the chain can be used to store the S_1^r-registers of $P_x(a, a, c, d, e)$ in successive forward sweeps. Each PE in the chain has duplicate S_1^r-registers to be used for this purpose. When $P_x(a, a, c, d, e)$ first becomes active at forward sweep a, it copies the contents of its S_1^r registers into the corresponding duplicate registers. In succeeding sweeps, the contents of the duplicate registers of every PE in the chain are shifted into the duplicate registers of the next PE in the chain. The new values of the S_1^r-registers of $P_x(a, a, c, d, e)$ can then be copied into its (now vacant) duplicate registers. These steps are illustrated in Fig. 5.2. The direction of shifting is simply reversed during regeneration, thus restoring the S_1^r-registers of $P_x(a, a, c, d, e)$ to their proper values. These values can then be reverse-routed to other PEs during the reverse sweep.

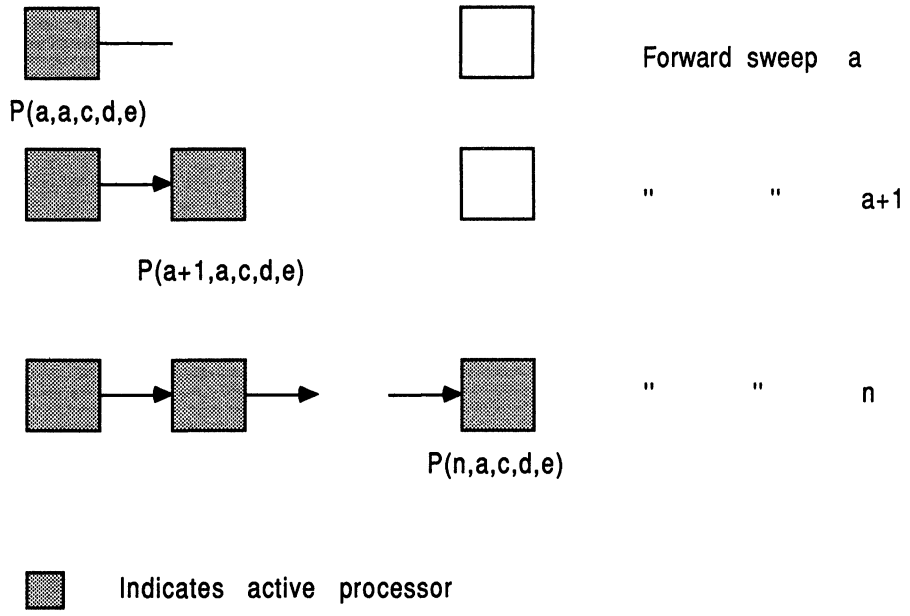


FIG. 5.2. *The chain of processors* $P(a, a, c, d, e)$, $P(a+1, a, c, d, e), \cdots, P(n, a, c, d, e)$ *is used to remember the contents of the* S_1' *registers of processor* $P(a, a, c, d, e)$ *at forward sweeps* a *to* n.

From the above discussion, it is clear that the data registers of all PEs are properly updated at each new reverse sweep. For primary PEs, we would also like that their A registers be properly updated. Moreover, for primary PEs computing B-items (i.e., PEs of the form $P_x(a, a, a, a, a)$), we would require updated values of their B registers. These can be accomplished by reversing the assignment statements labeled (1) and (2) in procedure *RECOGNIZE*, i.e.:

(1) If the PE is a primary PE, then $A \leftarrow eq(R1[0, 1].left) \cup eq(R1[1, 0].right) \cup eq(R2[0].right) \cup eq(R3[1].left)$.

(2) If the primary PE is of the form $P_x(a, a, a, a, a)$, then $B \leftarrow eq(R3[0].right)$ if $x = 0$, else $B \leftarrow eq(R2[1].left)$.

Finally, we observe that in regenerating forward sweep s from forward sweep $(s+1)$, PEs in subarray $P(s+1, *, *, *, *)$ should cease becoming active. This can be done by reversing the order of activations of PEs during the recognition phase (see § 4.4).

From the above discussion, we thus have the following.

THEOREM 5.1. *The configuration (contents of data registers, plus accumulator registers specified in* (1) *and* (2) *above) of each* PE *at reverse sweep* s, $0 \leqq s \leqq n$, *is identical to its configuration at the end of forward sweep* $(n - s)$.

5.2. The marking subphase. Concurrently with the regeneration phase, the processor array executes a marking subphase during which it searches and marks adjunction rules that make up a parse of the input string. In the following discussion, we assume that during each reverse sweep, the marking process is performed only after the regenerated values of the data registers have "settled."

Let $v = \langle conv, \Theta, \Gamma, \Delta \rangle$ be the rule found by processor $P(n, n, n, n, n)$ (in the B register of one of its PEs) at the end of the recognition phase. At reverse sweep 0, the marking subphase starts with $P(n, n, n, n, n)$ marking the rule v in the B registers of both its PEs. (In practice, a set of rules can be stored as a bit vector, one bit per rule; marking a rule then simply translates to marking the bit corresponding to the rule.)

Now, consider a processor of the form $P(a, a, a, a, a)$ and suppose that at reverse sweep $(n-s)$ it has a marked rule v in either of its B registers. One can verify that $v \in B(q, s)$, where $q = (s-a)$. During the same reverse sweep, $P(a, a, a, a, a)$ sends v to the sequence of processors $\{P(a, a, c, a, a) \mid c \leqq a\}$ via the $[0, 0, -1, 0, 0]$ links. Let P be the *first* processor in this sequence such that the A register of one of its PEs contains v. This processor then marks the occurrence of v in A and "disables" the search in the remaining processors by no longer sending v to the next processor in the sequence. Informally, the steps just described has the effect of "decomposing" $B(q, s)$ into its component items $\{A(q, r, r, s) \mid q \leqq r \leqq s\}$ and choosing one item containing v as the item from which a convolving pair of v is to be searched.

The marking process continues from every primary PE $P_x(a, b, c, b, b)$ that finds a marked rule v' in its A register. The PE first executes function *REDUCE* (v') (see § 2.3) to obtain a (possibly) new rule $v = \langle conv, \Theta, \Gamma, \Delta \rangle$. It then checks the *conv* field of v to determine the subarray of PEs from which to search a convolving pair of v. If $conv = 0$, then no search for a convolving pair is needed. If $conv = 2$, $P_x(a, b, c, b, b)$ sends v to PEs $\{P_x(a, b, c, d, b) \mid d \leqq b\}$. Let P be the first PE in this sequence such that for some register-pair $R2(y)$, there are rules $v_1 \in R2[y].left$ and $v_2 \in R2[y].right$ such that $v = v_1 *_2 v_2$. This PE marks *all* occurrences of v_1 (v_2) in $eq\,(R2[y].left)$ $(eq\,(R_2[y].right))$. The search is then disabled for the remaining PEs in the sequence. If $conv = 3$, the same steps are carried out except that v is sent to the subarray $\{P_x(a, b, c, b, e) \mid e \leqq b\}$ and the register-pairs examined are those in the set $\{R3[z]\}$.

The case when $conv = 1$ is slightly more complicated. Here, we wish to find exactly one processor in the *two-dimensional* subarray $\{P_x(a, b, c, d, e) \mid d, e \leqq b\}$ such that for some register-pair $R1[y, z]$ there are rules $v_1 \in R1[y, z].left$ and $v_2 \in R1[y, z].right$ such that $v = v_1 *_1 v_2$. A naive extension of the linear search described above to two dimensions would not work since convolving pairs in PEs the same distance away from $P_x(a, b, c, b, b)$ would be marked at the same time.

The problem can be solved by introducing the following modification to the recognition phase. While performing the union operation during a forward sweep, each PE in the "rightmost column" of the subarray (i.e., PEs $\{P_x(a, b, c, d, b) \mid d \leqq b\}$) collects from all PEs in the same row, the rules resulting from adjunctions $(conv = 1)$ and stores the set of rules in a temporary register, say *temp*. Each rightmost PE performs this step at every forward sweep. However, in order not to lose the values of *temp* during previous forward sweeps, the rightmost PE, say $P_x(a, b, c, d, b)$, shifts the previous values of *temp* along the chain of PEs $P_x(a, b, c, d, b)$, $P_x(a+1, b+1, c, d, b), \cdots, P_x(n, b+(n-a), c, d, b)$. (The steps are similar to those shown in Fig. 5.2.) This way, during the regeneration phase, the proper values of *temp* can be restored by performing the shifts in the reverse direction.

The two-dimensional search for a convolving pair can now be reduced to two linear searches as follows (see Fig. 5.3). From primary PE $P_x(a, b, c, b, b)$, the marked rule v is sent to the rightmost column of PEs in the subarray. Each rightmost PE receiving v first checks whether v is in register *temp*. If it is, then the rightmost PE "knows" that a convolving pair of v is in some PE in its row. The PE then disables the search in the succeeding rows and initiates a linear search for a convolving pair of v among the PEs in its own row, marking only the first convolving pair that is encountered.

The regeneration subphase that runs concurrently with the marking subphase eventually brings rules marked in the data registers to the A registers of primary PEs or the B registers of PEs of the form $P_x(a, a, a, a, a)$ (recall that these registers are

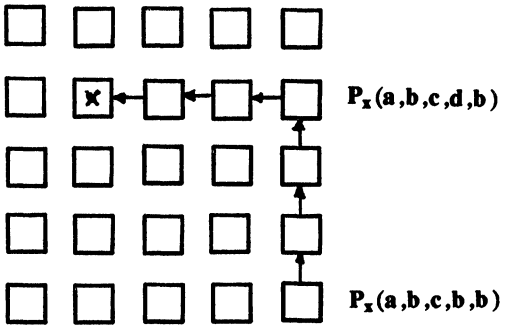


FIG. 5.3. *Two-dimensional search accomplished by two linear searches.*

updated as described in § 5.1). From these PEs the search-and-mark process is repeated. Thus, at the end of reverse sweep n (after all forward sweeps have been regenerated), all rules that make up a parse tree of the input string have been marked. From among these rules, only those indicating adjunctions ($conv = 1$) actually need be output. The outputting process is described in § 5.3.

5.3. The parse outputting subphase. The parse outputting subphase carries out the process of systematically outputting adjunction rules that are marked during the marking subphase. This subphase runs concurrently with the regeneration and marking subphases and is completed at the end of reverse sweep n.

Proposition 5.1 below follows directly from Proposition 2.1 and states the order in which the rules should be output.

PROPOSITION 5.1. *A parse of the input string can be obtained by outputting the adjunction rules marked during the marking subphase in the following order:*

(1) *Rules marked at reverse sweep s are output before those at reverse sweep $s-1$.*

(2) *For a fixed reverse sweep s, marked rules in subarray $P(a, *, *, *, *)$ are output before those in subarray $P(a+1, *, *, *, *)$.*

(3) *For a given reverse sweep and a fixed a, marked rules in subarray $P(a, b, *, *, *)$ are output before those in subarray $P(a, b+1, *, *, *)$.*

For example, consider Fig. 5.4, showing a sample parse tree for a string of length $n = 9$. The entries $A(i, j, k, l)$ shown in brackets next to the parse tree nodes (rules) indicate that the rule belongs to the corresponding matrix entry. Thus, for instance, rule R1 belongs to $A(1, 2, 4, 6)$. During the marking subphase, the rules would be marked in the PEs listed below the figure, at the specified reverse sweeps. According to Proposition 5.1, these rules are output in the order R2, R1, R3, R, that is also the order they are output by the sequential algorithm *SEARCH-FOR-PARSE* (see Proposition 2.1).

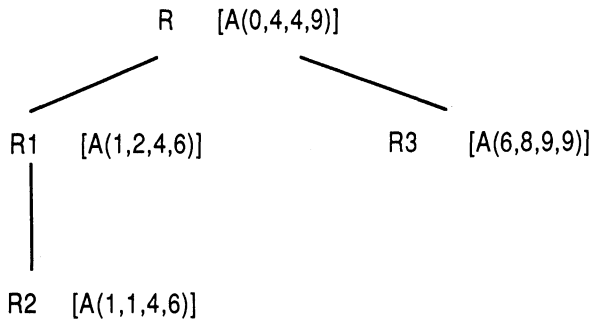
We now state the following fact that allows us to perform the routing required to output the rules in the desired order.

FACT 5.3. In any reverse sweep and for fixed a and b:

(1) There is at most one marked adjunction rule in the subarray of primary processors $P(a, b, *, b, b)$.

(2) If there is a marked adjunction rule in subarray $P(a, b, *, b, b)$, then there are no marked adjunction rules in subarrays $\{P(a', b', *, b', b') \mid a' < a, b' \geqq b\}$ and $\{P(a', b', *, b', b') \mid a' > a, b' \leqq b\}$.

Fact 5.3(1) implies that if in any reverse sweep s there is a marked adjunction rule in primary processor $P(a, b, c, b, b)$, this rule can be sent to processor $P(a, b, 0, b, b)$



Sample parse: R_2, R_1, R_3, R.

R∈[A(0,4,4,9)] is in $P_0(9,9,5,9,9)$ at reverse sweep 0

R1∈[A(1,2,4,6)] is in $P_0(5,3,2,3,3)$ at reverse sweep 3

R2∈[A(1,1,4,6)] is in $P_0(5,2,2,2,2)$ at reverse sweep 3

R3∈[A(6,8,9,9)] is in $P_1(3,2,2,2,2)$ at reverse sweep 0

FIG. 5.4. *A sample parse tree for a string of length* $n = 9$.

via the $[0, 0, -1, 0, 0]$ links without "colliding" with any other rule (since there is at most one).

Now consider the rules reaching processors $\{P(a, b, 0, b, b) | 0 \leqq b \leqq a \leqq s\}$ at reverse sweep s. By Fact 5.3(2), the rules in these processors can only have the pattern shown in Fig. 5.5. In particular, if the processor labeled "*" contains a marked

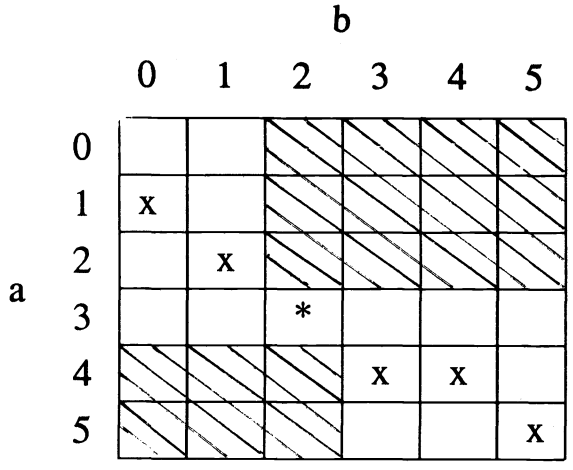


FIG. 5.5. *Interpretation of Fact* 5.3(2).

adjunction rule, then there can be no marked rules in the shaded areas. This implies that there can be at most one rule in any column of the subarray and the rule, when read from left to right, satisfies Proposition 5.1(2) and (3).

The following steps can thus be performed. At each reverse sweep s, every processor $P(a, b, 0, b, b)$ containing a marked adjunction rule sends the rule to processor $P(0, b, 0, b, b)$ via the $[-1, 0, 0, 0, 0,]$ links, then to processor $P(0, b, 0, 0, 0)$ via the $[0, 0, 0, -1, -1]$ links. Processor $P(0, b, 0, 0, 0)$ in turn outputs any marked adjunction rule it receives. The rules output by the linear subarray $P(0, b, 0, 0, 0) | 0 \leqq b \leqq n\}$ thus satisfies Proposition 5.1(2) and (3) when read off starting from $P(0, 0, 0, 0, 0)$ to $P(0, n, 0, 0, 0)$.

For example, in the case of the parse tree of Fig. 5.4, rule R3, which is marked in processor $P(3, 2, 2, 2, 2)$ at reverse sweep 0, is first routed $P(3, 2, 0, 2, 2)$, then to $P(0, 2, 0, 2, 2)$, and finally to processor $P(0, 2, 0, 0, 0)$ where it is output. The other rules are routed in a similar way. Fact 5.3 guarantees that each rule reach its final destination without colliding with any other rule.

Finally, if the outputs of the subarray $\{P(0, b, 0, 0, 0) | 0 \leqq 0 \leqq n\}$ are listed in reverse order of sweeps, (i.e., insert the rules output in reverse sweep s before those output in reverse sweep $s-1$), the resulting sequence of rules constitutes a complete parse of the input string (see Proposition 5.1(1)).

Remark 5.1. With an additional number of linear steps, the rules output by the subarray can in fact be pipelined so that they are output in the desired order from processor $P(0, 0, 0, 0, 0)$. Intuitively, this can be done because although we are observing $O(n^2)$ outputs from the subarray, there are at most n marked adjunction rules that actually appear and the rest are dummy outputs. We omit the details, as the technique is similar to that described in [CHAN87].

Counting the time for the recognition phase, the total time complexity of the parallel algorithm is easily shown to be $(12n+1) = O(n)$, where n is the length of the input string.

6. Conclusion. We have presented an optimal linear-time parallel parsing algorithm for tree adjoining languages, a class that properly includes all context-free languages. The parallel model is quite simple in that it consists of finite-state processors whose function and size are independent of the length of the input.

We also mention that TALs can be shown to be $NC^{(2)}$ (= class of languages recognizable by $O(\log^2 n)$-time bounded and $O(\log n)$-space bounded alternating Turing machines (ATMs)) by extending the CFL recognition algorithm on an ATM given in [RUZZ80]. However, just like the latter, converting the ATM to a uniform Boolean circuit or a PRAM results in an inordinate increase in the number of processors (n^{12} processors in $O(\log^2 n)$ time seems the best so far). The same is true in fact for CFLs for which the best known PRAM algorithm operates in $O(\log^2 n)$ time using n^6 processors [RYTT85].

It is thus an interesting open question whether TALs (or even CFLs) can be recognized in sublinear parallel time using a near-optimal number of processors.

7. Appendix. In this appendix we sketch the procedure for deriving the routing table (Table 4.1), and show that procedures *UPDATE-REGISTERS* and *ROUTE-REGISTERS* (that make use of the table) properly update and route the values of data registers of every active PE. To do this, it is convenient to state the "inverse mapping" of Lemma 4.1.

DEFINITION. For integers s, a, b, c, d, e, and $x, y, z \in \{0, 1\}$, define

$$I_x = s - a, x \in \{0, 1\},$$

$$J_x = \begin{cases} s-a+b-c & \text{if } x=0, \\ s-a+c & \text{if } x=1, \end{cases}$$

$$K_x = \begin{cases} s-c & \text{if } x=0, \\ s-b+c & \text{if } x=1, \end{cases}$$

$$L_x = s,\ x \in \{0,1\},$$

$$M_{xy} = \begin{cases} s-a+b-d & \text{if } x \in \{0,1\} \text{ and } y=0, \\ s-a-c+d & \text{if } x=0 \text{ and } y=1, \\ s-a-b+c & \text{if } x=1 \text{ and } y=1, \end{cases}$$

$$P_{xz} = \begin{cases} s+b-c-e & \text{if } x=0 \text{ and } z=0, \\ s+c-e & \text{if } x=1 \text{ and } z=0, \\ s-b+e & \text{if } x \in \{0,1\} \text{ and } z=1. \end{cases}$$

LEMMA A1 (Inverse Mapping of Lemma 4.1). *At forward sweep s, the data registers of an active* PE $P_x(a, b, c, d, e)$ *have the following values*:

(1) $R1[y, z].left = A(M_{xy}, J_x, K_x, P_{xz})$,
(2) $R1[y, z].right = A(I_x, M_{xy}, P_{xz}, L_x)$,
(3) $R2[y].left = B(I_x, M_{xy})$,
(4) $R2[y].right = A(M_{xy}, J_x, K_x, L_x)$,
(5) $R3[z].left = A(I_x, J_x, K_x, P_{xz})$,
(6) $R3[z].right = B(P_{xz}, L_x)$.

DEFINITION. Link $[\Delta a, b, \Delta c, \Delta d, \Delta e]$ ($[\Delta a, \Delta b, \Delta c, \Delta d, \Delta e]^+$) is said to *cover* data register r of PE $P_x(a, b, c, d, e)$ if and only if the value stored in this register is the same as that stored in register r of $P_x(a-\Delta a, b-\Delta b, c-\Delta c, d-\Delta d, e-\Delta e)$ at forward sweep s $(s-1)$.

The routing table (Fig. 4.1) is essentially a list of links that cover the data registers of a PE. We now show how this can be derived. We only illustrate the derivation for data registers $R1[y, z].left$; the links covering other data registers can be obtained similarly.

Consider data registers $R1[y, z].left$ of an active PE $P_x(a, b, c, d, e)$, such that x, y, $z \in \{0, 1\}$. Suppose that $x = y = z = 0$. Then by Lemma A1, at forward sweep $s \geqq a$, register $R1[0, 0].left$ of $P_0(a, b, c, d, e)$ contains $A(M_{00}, J_0, K_0, L_0)$. We wish to find an active PE $P_0(a-\Delta a, b-\Delta b, c-\Delta c, d-\Delta d, e-\Delta e)$ such that at forward sweep $s-\Delta s$, register $R1[0, 0].left$ of this PE also contains $A(M_{00}, J_0, K_0, L_0)$. By Lemma A1, it should be the case that

$$M_{00} = s-a+b-d = (s-\Delta s)-(a-\Delta a)+(b-\Delta b)-(d-\Delta d),$$
$$J_0 = s-a+b-c = (s-\Delta s)-(a-\Delta a)+(b-\Delta b)-(c-\Delta c),$$
$$K_0 = s-c = (s-\Delta s)-(c-\Delta c),$$
$$P_{00} = s+b-c-e = (s-\Delta s)+(b-\Delta b)-(c-\Delta c)-(e-\Delta e),$$

or equivalently,

$$\begin{aligned} \Delta s - \Delta a + \Delta b \quad\quad - \Delta d \quad\quad &= 0, \\ \Delta s - \Delta a + \Delta b - \Delta c \quad\quad &= 0, \\ \Delta s \quad\quad - \Delta c \quad\quad &= 0, \\ \Delta s \quad\quad + \Delta b - \Delta c \quad\quad - \Delta e &= 0, \end{aligned}$$

where $\Delta s, \cdots, \Delta e \in \{0, 1\}$ but not all zero.

The solutions $(\Delta s, \Delta a, \Delta b, \Delta c, \Delta d, \Delta e)$ to the above system of equations are $(1, 1, 1, 1, 1, 1)$, $(1, 0, 0, 1, 1, 0)$, and $(0, 1, 1, 0, 0, 1)$ that correspond to links

$[1, 1, 1, 1, 1]^+$, $[0, 0, 1, 1, 0]^+$, and $[1, 1, 0, 0, 1]$, respectively (recall that a superscript "+" denotes a sweep delay).

Each of the above links covers register $R1[0, 0].\mathit{left}$ of an active PE $P_0(a, b, c, d, e)$ only if the adjacent PE associated with the link is also active (otherwise, the latter by definition cannot route items stored in its registers). We now determine under what conditions this property holds.

Consider first link $[1, 1, 1, 1, 1]^+$ that represents data being forwarded from PE $P_0(a-1, b-1, c-1, d-1, e-1)$ at forward sweep $s-1$ to PE $P_0(a, b, c, d, e)$ at forward sweep s. If both PEs are to be active, then by Fact 4.2 it should be the case that

(1) $(b \leqq a \leqq s) \Rightarrow (b-1 \leqq a-1 \leqq s-1)$,

(2) $(b \leqq 2c \leqq 2b) \Rightarrow (b-1 \leqq 2c-2 \leqq 2b-2)$,

(3) $(b+c \leqq 2d \leqq 2b) \Rightarrow (b+c-2 \leqq 2d-2 \leqq 2b-2)$, and

(4) $2b-c \leqq 2e \leqq 2b) \Rightarrow (2b-c-1 \leqq 2e-2 \leqq 2b-2)$.

It is easy to see that conditions (1) and (3) are always true. On the other hand, (2) is true except when $2c = b$ and (4) is true except when $2e = 2b - c$. It follows that link $[1, 1, 1, 1, 1]^+$ covers register $R1[0, 0].\mathit{left}$ of PE $P_0(a, b, c, d, e)$ except when $2c = b$ or $2e = 2b - c$.

Similarly, it can be shown that link $[0, 0, 1, 1, 0]^+$ covers the register except when $s = a$ or $2c = b$, $b+1$ or $2d = b + c$ or $2e = 2b - c$. Observe, however, that anything covered by this link is also covered by $[1, 1, 1, 1, 1]^+$. Thus, $[0, 0, 1, 1, 0]^+$ is redundant and can be eliminated. Finally, $[1, 1, 0, 0, 1]$ covers the register except when $b = c$ or $b = d$. Thus, the two links $\{[1, 1, 1, 1, 1]^+, [1, 1, 0, 0, 1]\}$ together cover the register except when ($2c = b$ or $2e = 2b - c$) and ($b = c$ or $b = d$), or equivalently, ($b = c = 0$) or ($2c = b$ and $b = d$) or ($2e = 2b - c$ and ($b = c$ or $b = d$)).

Proceeding in exactly the same manner, one can determine the links covering all other data registers $R1[y, z].\mathit{left}$ of $P_x(a, b, c, d, e)$ for other values of x, y, and z, and under what conditions the coverings hold. These are summarized in Table A1. Observe that (except for the last column) Table A1 is exactly the routing table given in Table 4.1.

The exceptions in the last column of Table A1 imply that certain data registers of certain PEs may in fact be not covered by any of their associated links. However, recall that procedure *UPDATE-REGISTERS* operates in such a way that any item targeted for data register r of PE P is placed in all registers $r' \in eq(P, r)$. Thus, it is sufficient to show that for any data register, there is some register in the same equivalence class that is covered by some link. The register equivalence classes are given in Fact 4.1, which we restate here for quick reference.

FACT 4.1. Let $P_x(a, b, c, d, e)$ be a PE. Then at any forward sweep:

(1) If ($x = 0$ and $2d = b + c$) or ($x = 1$ and $2d = 2b - c$), $R1[0, z].t = R1[1, z].t$ and $R2[0].t = R2[1].t$, where $t \in \{\mathit{left}, \mathit{right}\}$.

(2) If ($x = 0$ and $2e = 2b - c$) or ($x = 1$ and $2e = b + c$), $R1[y, 0].t = R1[y, 1].t$ and $R3[0].t = R3[1].t$, where $t \in \{\mathit{left}, \mathit{right}\}$.

(3) If $2c = b$, every data register r of $P_x(a, b, c, d, e)$ has the same contents as data register r of $P_{\bar{x}}(a, b, c, d, e)$.

For convenience, we shall call a PE satisfying condition (i) above as a type (i) PE.

First observe from Table A1 that regardless of its type, an active PE $P_x(a, b, c, d, e)$ satisfying $b = c = 0$ does not have a covered register. However, since $d, e \leqq b$ these PEs are necessarily of the form $P_x(a, 0, 0, 0, 0)$ and hence are PEs that compute boundary items (see procedure *RECOGNIZE*) and do not require data from adjacent PEs. Similarly, Table A1 states that register $R1[0, 1].\mathit{left}$ of any PE satisfying $b = d = e$ (a primary PE) is not covered. However, for a primary PE, $R1[0, 1].\mathit{left}$ is the register

TABLE A1
Links covering registers $R1[x, y].left$ *of* PE $P_x(a, b, c, d, e)$.

$P_0(a, b, c, d, e)$		
Register	Links	Exceptions
$R1[0, 0] \cdot left$	$[1, 1, 1, 1, 1]^+$ $[1, 1, 0, 0, 1]$	$(b = c = 0) + 2 \cdot (b = c) + 2 \cdot (b = d) + 3 \cdot (b = d)$
$R1[0, 1] \cdot left$	$[1, 1, 1, 1, 0]^+$ $[1, 1, 0, 0, 1]$	$(b = c = 0) + (b = d = e) + 3 \cdot (b = d)$
$R1[1, 0] \cdot left$	$[1, 1, 1, 1, 1]^+$ $[1, 1, 0, 1, 1]$	$(b = c = 0) + 1 \cdot 2 + 1 \cdot 3 + 2 \cdot (b = c)$
$R1[1, 1] \cdot left$	$[1, 1, 1, 1, 0]^+$ $[1, 1, 0, 1, 1]$	$(b = c = 0) + 1 \cdot 3 + 1 \cdot (b = e)$
$P_1(a, b, c, d, e)$		
Register	Links	Exceptions
$R1[0, 0] \cdot left$	$[1, 1, 0, 1, 1]^+$ $[1, 1, 1, 0, 1]$	$(b = c = 0) + 2 \cdot 3 + 2 \cdot (b = d)$
$R1[0, 1] \cdot left$	$[1, 1, 0, 1, 0]^+$ $[1, 1, 1, 0, 1]$	$(b = c = 0) + (b = d = e) + 3 \cdot (b = e)$
$R1[1, 0] \cdot left$	$[1, 1, 0, 1, 1]^+$ $[1, 1, 1, 1, 1]$	$(b = c = 0) + 1 \cdot 2 + 1 \cdot (b = c) + 2 \cdot 3$
$R1[1, 1] \cdot left$	$[1, 1, 0, 1, 0]^+$ $[1, 1, 1, 1, 1]$	$(b = c = 0) + 1 \cdot (b = c) + 1 \cdot (b = e) + 3 \cdot (b = e)$

Legend:
$1 \equiv (x = 0$ and $b + c = 2d)$ or $(x = 1$ and $2b - c = 2d)$
$2 \equiv (x = 0$ and $2b - c = 2e)$ or $(x = 1$ and $b + c = 2e)$
$3 \equiv (2c = b)$

assigned to the A-item computed by this PE during the sweep and hence should not be covered.

The rest of the exceptions listed in Table A1 can be handled by considering PEs of different types and applying Fact 4.1. For instance, suppose that the PE is type (1). Then, from Table A1, registers to $R1[0, z].left$, $z \in \{0, 1\}$, of the PE are not covered. However, by Fact 4.1(1), $R1[1, z].left$ is equivalent to $R1[0, z].left$, which is covered. Since *UPDATE-REGISTERS* places the value targeted for a register in all registers in the same equivalence class, it is clear that for a type (1) PE, registers $R1[1, z].left$ would also be updated. A similar analysis can be carried out for other PE types.

To summarize, using Fact 4.1, the following proposition can be proved.

PROPOSITION A1. *Let* $P = P_x(a, b, c, d, e)$ *be an active* PE. *For every data register* r *of* P, *there is at least one register* $r' \in eq(P, r)$ *such that some link in the routing table covers* r', *except for the following cases*:

(1) *If* P *is such that* $b = c = d = e = 0$, *then no data register of* P *is covered.*

(2) *If* P *is such that* $b = d = e$, *then data registers in the sets* $eq(R1[0, 1].left)$, $eq(R1[1, 0].right)$, $eq(R2[0].right)$, *and* $eq(R3[1].left)$ *are not covered.*

(3) *If* P *is such that* $a = b = c = d = e$, *then data registers in the sets* $eq(R3[0].right)$ *(if* $x = 0$*) or* $eq(R2[1].left)$ *(if* $x = 1$*) are not covered.*

Part (1) corresponds to PEs computing boundary items; such PEs do not receive data from adjacent PEs. Part (2) corresponds to primary PEs; the value stored in the registers listed is the A-item computed within the PE during the sweep. Finally, (3) corresponds

to PEs of the form $P_x(a, a, a, a, a)$ that compute B-items; the value stored in the registers listed is the B-item computed within the PE. (See procedure *RECOGNIZE* for details.)

Proposition A1 implies that the procedures *UPDATE-REGISTERS* and *ROUTE-REGISTERS* properly updates and routes the data registers of every active PE. We also mention the fact that *ROUTE-REGISTERS* may actually forward items from an active PE to an inactive PE. However, since by definition an inactive PE does not participate in any computation or data routing, items received by inactive PEs are simply discarded.

REFERENCES

[AHO72] A. V. AHO AND J. D. ULLMAN, *The Theory of Parsing, Translation and Computing*, Vol. 1, Prentice Hall, Englewood Cliffs, NJ, 1972.

[CHAN87] J. H. CHANG, O. H. IBARRA, AND M. A. PALIS, *Parallel parsing on a one-way array of finite-state machines*, IEEE Trans. Comput., 36 (1987), pp. 64–75.

[CHIA84] Y. CHIANG AND K. S. FU, *Parallel parsing and* VLSI *implementations for syntactic pattern recognition*, IEEE Trans. Pattern Anal. Machine Intelligence, 6 (1984), pp. 302–313.

[COPP87] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication for arithmetic progressions*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 1–6.

[HOPC79] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.

[JOSH75] A. K. JOSHI, L. S. LEVY, AND M. TAKAHASHI, *Tree adjunct grammars*, J. Comput. System Sci., 10 (1975), pp. 136–163.

[KOSA75] S. R. KOSARAJU, *Speed of recognition of context-free languages by array automata*, SIAM J. Comput., 4 (1975), pp. 331-340.

[KROC85] A. S. KROCH AND A. K. JOSHI, *The linguistic relevance of tree adjoining grammars*, Technical Report MS-CIS-85-16, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, April 1985.

[RYTT85] W. RYTTER, *The complexity of two-way pushdown automata and recursive programs*, in Combinatorial Algorithms on Words, A. Apostolico and Z. Galil, eds., Springer-Verlag, Heidelberg, 1985, pp. 341–356.

[RUZZ80] W. L. RUZZO, *Tree-size bounded alternation*, J. Comput. Systems Sci., 22 (1980), pp. 218–235.

[VALI75] L. VALIANT, *General context-free recognition in less than cubic time*, J. Comput. System Sci., 10 (1975), pp. 308-315.

[VIJA86] K. VIJAYASHANKER AND A. K. JOSHI, *Some computational properties of tree adjoining grammars*, in Proc. 11th Meeting of Association of Computational Linguistics, University of Chicago, Chicago, IL, August 1986.

[VIJA87] K. VIJAYASHANKER, *Tree adjoining grammars*, Ph.D. dissertation, Department of Computer and Information Science, University of Pennsylvania, Philadelphia, PA, 1987.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 32-43, February 1990

002

COMPLEXITY AND UNSOLVABILITY PROPERTIES OF NILPOTENCY*

I. R. HENTZEL† AND D. POKRASS JACOBS‡

Abstract. A nonassociative algebra is nilpotent if there is some n such that the product of any n elements, no matter how they are associated, is zero. Several related, but more general, notions are left nilpotency, solvability, local nilpotency, and nillity. First the complexity of several decision problems for these properties is examined. In finite-dimensional algebras over a finite field it is shown that solvability and nilpotency can be decided in polynomial time. Over Q, nilpotency can be decided in polynomial time, while the algorithm for testing solvability uses a polynomial number of arithmetic operations, but is not polynomial time. Also presented is a polynomial time probabilistic algorithm for deciding left nillity. Then a problem involving algebras given by generators and relations is considered and shown to be NP-complete. Finally, a relation between local left nilpotency and a set of natural numbers that is 1-complete for the class Π_2 in the arithmetic hierarchy of recursion theory is demonstrated.

Key words. nonassociative algebra, nilpotent, solvable, NP-complete, power associative, arithmetic hierarchy, recursively enumerable, recursive, 1-complete

AMS(MOS) subject classifications. 68Q25, 68Q15, 17A99, 03D55

1. Introduction.

A nonassociative algebra (or simply an algebra) over a field F is a set A together with two binary operations, $*$ and $+$, such that $(A, +)$ is a vector space over F,

$$x*(y+z)=x*y+x*z, \tag{1}$$

$$(y+z)*x=y*x+z*x, \tag{2}$$

$$\alpha(x*y)=(\alpha x)*y=x*(\alpha y) \tag{3}$$

for all $x, y, z \in A$, and $\alpha \in F$. The operation $*$ is not necessarily associative. Throughout this paper we shall suppress the $*$ by writing, for example, xy instead of $x*y$.

If B and C are arbitrary sets in a nonassociative algebra A, then by BC we usually mean the subspace spanned by all elements in $\{bc \mid b \in B, c \in C\}$.

For each integer $n \geqq 1$ let us denote by A^n the subspace spanned by all products of n (not necessarily distinct) elements in A, in all $(1/n)\binom{2n-2}{n-1}$ associations. Let us now define $A^{(1)}=A^{\langle 1\rangle}=A^{[1]}=A$. We then define, for each $n \geqq 1$,

$$A^{(n+1)}=A^{(n)}A^{(n)}, \qquad A^{\langle n+1\rangle}=A^{\langle n\rangle}A+AA^{\langle n\rangle}, \qquad A^{[n+1]}=AA^{[n]}.$$

It is clear that the A^n and $A^{\langle n\rangle}$ are descending chains of ideals, the $A^{[n]}$ is a descending chain of left ideals, and the $A^{(n)}$ is a descending chain of subalgebras.

If for some n, $A^n=\{0\}$, we say A is *nilpotent.* In this case, the minimal such n for which A^n is zero is the *index* of nilpotency. If $A^{[n]}=\{0\}$, A is *left nilpotent,* and if $A^{(n)}=\{0\}$, A is *solvable.* (The reader will note that, unfortunately, in this paper "solvability" carries two quite different meanings, one being the above algebraic definition and the other being from logic.) The index of left nilpotency and index of solvability are defined in an analogous manner to the index of nilpotency. It is easy to see that for all $i \geqq 1$

$$A^{(i)} \subseteq A^{[i]} \subseteq A^i$$

* Received by the editors January 20, 1988; accepted for publication (in revised form) March 9, 1989.
† Department of Mathematics, Iowa State University, Ames, Iowa 50011.
‡ Department of Computer Science, Clemson University, Clemson, South Carolina 29634-1906.

and so nilpotency implies left nilpotency, which implies solvability. It is also easy to show that for each $i \geqq 1$,

$$A^{2^{i-1}} \subseteq A^{\langle i \rangle} \subseteq A^i. \tag{4}$$

The second containment in (4) follows by induction on i since $A^{\langle i \rangle} = A^{\langle i-1 \rangle}A + AA^{\langle i-1 \rangle} \subseteq A^{i-1}A + AA^{i-1} \subseteq A^i$. The first containment in (4), namely $A^{2^{i-1}} \subseteq A^{\langle i \rangle}$, is also obtained by induction on i. When $i = 1$, in fact, equality holds. Assuming we have $A^{2^{i-1}} \subseteq A^{\langle i \rangle}$ for some $i \geqq 1$, consider now any $x \in A^{2^i}$. Then x is a linear combination of a finite number of products pq where $p \in A^s$ and $q \in A^t$, for some s and t in which $s + t = 2^i$. Either $s \geqq 2^{i-1}$ or $t \geqq 2^{i-1}$. In the first case, we have $pq \in A^s A^t \subseteq A^{2^{i-1}}A \subseteq A^{\langle i \rangle}A \subseteq A^{\langle i+1 \rangle}$. In the second case we have $pq \in AA^{\langle i \rangle} \subseteq A^{\langle i+1 \rangle}$, and hence $A^{2^i} \subseteq A^{\langle i+1 \rangle}$, completing the proof.

An algebra is called *power associative* if each element generates an associative subalgebra. In such an algebra we say an element x is *nil* if there exists some k, depending on x, such that $x^k = 0$. A power associative algebra is called nil if each member is nil. This is equivalent to saying that the subalgebra generated by x is nilpotent. Note that in power associative algebras, nillity is implied by solvability, and hence by nilpotency and left nilpotency.

Nilpotency and its related properties are important to the theory of algebras, since the radical of an algebra, under suitable conditions, is nilpotent. These properties have received thorough mathematical investigation.

The theory of complexity has been applied to both associative and nonassociative algebras [4], [9]. The purpose of this paper is to investigate the complexity and degree of unsolvability of certain questions about nilpotency. Our paper is organized so that the computationally easier questions are studied first. For example, the next section deals with questions about finite-dimensional algebras that can be answered by algorithms performing a number of operations polynomial in the dimension of the algebra. We then consider a probabilistic approach to nillity and nilpotency. The next section deals with an NP-complete question. Finally, our last section classifies an unsolvable problem by proving it to be 1-complete for the class Π_2 in the arithmetic hierarchy of recursion theory.

2. Polynomially answerable questions. In this section all algebras are assumed to be finite-dimensional over a fixed field F. We assume that F is either a finite field or is Q, the field of rationals.

Given a finite-dimensional algebra A over F, multiplication on A is usually described by specifying a basis $v_1, \cdots, v_n$ and then giving a table that gives the product of any two basis elements. The table consists of all δ_{ijk} such that $v_i v_j = \sum_{k=1}^{n} \delta_{ijk} v_k$. The δ_{ijk}'s are called *structure constants.* Given these, the multiplication of two vectors from A can be computed by applying laws (1), (2), and (3) to arbitrary linear combinations of the basis vectors.

Each instance of an n-dimensional algebra is therefore encoded by a sequence of n^3 constants. In the case of an algebra over Q we assume that the rational constants are given as pairs of relatively prime integers. Now let P be an algorithm for solving a decision problem for finite-dimensional algebras encoded in this way. If x is a string that encodes an algebra we let $|x|$ denote the length of x. Usually the complexity of P is measured as a function $T(n)$, indicating the maximum running time of P over all encodings of length n. Initially, in the algorithms described in Theorems 1 and 2 below, we deviate from this approach in two ways.

First, for the two algorithms that follow, we measure their running times as a function of the dimension of the algebra A. Note that for algebras over a finite field,

$|x|$ is always $O(n^3)$ where n is the dimension of the algebra that x encodes. However for Q this is not true since arbitrarily long strings can encode algebras of the same dimension.

Second, we initially calculate the running time of an algorithm by estimating the number of arithmetic operations (addition, subtraction, multiplication, or division) that occur in F. We caution that this is somewhat misleading since, as we will see, over Q it is possible for an algorithm to perform only a polynomial number of operations, and yet require an exponential amount of time as a function of its input length. However when we use the term *polynomial time* we use it in its usual sense and we will carefully distinguish between a polynomial time algorithm and one for which merely the number of arithmetic operations is polynomial.

Let us now make the following observations. If v_i and v_j are basis elements of an n-dimensional algebra, and $\beta \in F$, then computing $\beta v_i v_j$ takes n multiplications in F. If w is an arbitrary vector, say $w = \sum_{i=1}^{n} \alpha_i v_i$, then multiplying $v_j w = \sum_{i=1}^{n} \alpha_i v_j v_i$ takes n^2 multiplications and at most n^2 additions, and so $O(n^2)$ operations in all. Finally, if $u = \sum \alpha_i v_i$ and $w = \sum \beta_j v_j$ are arbitrary vectors then $uw = \sum_{i=1}^{n} \alpha_i v_i w$ takes $O(n^3)$ arithmetic operations. It follows that, in general, k arbitrary vectors can be multiplied, regardless of the association, by using $O((k-1)n^3)$ operations.

The algorithm described below will serve as a template throughout this section.

THEOREM 1. *Solvability of an n-dimensional algebra A can be decided using* $O(n^6)$ *arithmetic operations.*

Proof. Assume A is solvable. Then if $A^{(i)} \neq \{0\}$ we must have $A^{(i+1)}$ properly contained in $A^{(i)}$ and so $\dim(A^{(i+1)}) < \dim(A^{(i)})$. Hence an n-dimensional algebra is solvable if and only if $A^{(n+1)} = \{0\}$. The following algorithm computes a basis B for $A^{(n+1)}$. As the loop finishes each iteration for $i = 2, \cdots, n+1$, a basis for $A^{(i)}$ has been found.

Initialize B to the basis for A.
for $i := 2$ to $n+1$
(1) Let $C = \{bb' \mid b \text{ and } b' \in B\}$.
(2) Redefine B to be a basis for the span of C.

Note step (1) in the loop involves at most n^2 multiplications of arbitrary vectors. Since each such vector multiplication takes $O(n^3)$ operations, step (1) takes $O(n^5)$ operations. In step (2) the basis for C can be found by reducing an $m \times n$ matrix, where $m \leqq n^2$, to row canonical form. This can be done with $O(n^4)$ operations, and so step (1) dominates the loop. Since the loop iterates n times, the algorithm performs $O(n^6)$ operations. □

THEOREM 2. *Nilpotency of an n-dimensional algebra can be decided in* $O(n^5)$ *operations.*

Proof. By relation (4), an algebra is nilpotent if and only if $A^{\langle i \rangle} = \{0\}$ for some i. By a dimensionality argument similar to the one used in the case of solvability, it suffices to compute a basis B for $A^{\langle n+1 \rangle}$. An algorithm to compute B can be obtained from the previous algorithm by replacing its step (1) with

(1) Let $C = \{bv_i, v_i b \mid b \in B\}$

where the v_i are assumed to be the original basis elements of A. The construction of C involves the multiplication of at most $2n^2$ vectors. This time, however, the multiplication involves a full vector b with a basis element v_i, a process taking only $O(n^2)$ field operations and so constructing C takes $O(n^4)$. The remainder of the analysis is similar to that of the previous algorithm. □

There are some important distinctions between the two algorithms presented so far. First, the algorithm of Theorem 1 can be easily modified to compute the *index* of solvability of A. Indeed the algorithm need only check for the first i for which C is zero. On the other hand, while the algorithm of Theorem 2 can be modified in a similar way to detect the minimal i for which $A^{(i)} = \{0\}$, it apparently *cannot tell* the minimal i for which $A^i = \{0\}$.

A second distinction between the two algorithms concerns their computational complexity. As noted earlier, for finite fields the length of an input string x is always $O(n^3)$ where n is the dimension of the algebra that x encodes. Furthermore, for a finite field the operations each are bounded by a constant amount of time. Since the algorithms of Theorems 1 and 2 are dominated by the time spent performing arithmetic operations, Theorem 3 follows.

THEOREM 3. *Over a finite field, nilpotency and solvability can be decided in polynomial time. In the latter case, the index of solvability can also be calculated in polynomial time.*

Let us now consider what happens in the above algorithms when $F = Q$. In the first algorithm, testing for solvability is performed by repeatedly "squaring" the subalgebra $A^{(i)}$, n times. Since squaring a number (represented in base 2, say) approximately doubles the length of its representation, structure constants of length k can produce coefficients of about length $k2^n$. The following simple examples illustrate this. For a given n, consider the n-dimensional algebra with basis $v_1, v_2, \cdots, v_n$ where $v_1 v_2 = 2v_1$ with all other products zero. The first algorithm is easily seen to experience exponential growth since the encodings of its numbers become exponentially long.

Next, consider the algorithm of Theorem 2. We will prove that for $F = Q$ its complexity *is* bounded by a polynomial in the length of its input. However first let us make the following simplifying observation. If A is an algebra having structure constants $\{\delta_{ijk}\}$ and $0 \neq c \in Q$, define A_c to be the algebra with the same basis, but having structure constants $\{c\delta_{ijk}\}$. It is easy to show that A is nilpotent if and only if A_c is nilpotent. Consequently, by multiplying all structure constants of A by a common multiple of their denominators, we achieve an algebra A_c in which all structure constants are integers. If x is a string which encodes A and x_c is a string encoding A_c, then we have $|x_c| \leqq |x|^2$, and so there is no loss in assuming all algebras have integer structure contents.

If w is a vector in an algebra over Q with basis $\{v_1, \cdots, v_n\}$, and $w = \sum_{i=1}^{n} n_i v_i$, where the n_i are integers, let us agree that $\|w\| = \max\{|n_i|\}$, the largest absolute value of the coefficients.

LEMMA 1. *Let A be an n-dimensional algebra over Q with integer structure constants $\{\delta_{ijk}\}$ and let $m = \max\{|\delta_{ijk}|\}$. Let u be the product of k factors, each a basis element. Then $\|u\| \leqq n^{2(k-1)} m^{k-1}$.*

Proof. This follows immediately from two observations: First, for each basis element v_i, $\|v_i\| = 1$. Second, for arbitrary vector u and w we have $\|uw\| \leqq \|u\| \cdot \|w\| \cdot n^2 \cdot m$. □

THEOREM 4. *For finite-dimensional algebras over Q, nilpotency can be decided in polynomial time.*

Proof. We claim that the algorithm of Theorem 2 runs in time polynomial in its input length when $F = Q$. Let A be an algebra represented by a string x. If A has dimension n, then $n < |x|$. By the observation above we may assume all structure constants of A are integers. Let max be the largest absolute value of all such integers. The algorithm repeatedly executes steps (1) and (2), $n < |x|$ times. Therefore it suffices to show each of these steps requires only a polynomial in $|x|$ amount of time. Note that the members of C in step (1) are all products of at most $n+1$ basis elements. By

Lemma 1 the absolute value of their coefficients are bounded by $n^{2n}\max^n$ and hence have *length*

$$O(\log(n^{2n}\max^n)) = O(n\log(n)+n\log(\max)) \leqq O(|x|^2).$$

Note that step (2) need not change the values of coefficients. It can be accomplished by selecting a maximal linearly independent subset of C. This can be done by considering the members of C as rows and forming the matrix of all such rows. We then reduce this matrix to a "row canonical form," but without interchanging any rows. The nonzero rows that remain will be a basis for the row space. Moreover the original rows corresponding to these nonzero rows will also be a basis. This requires applying Gaussian elimination to an $m \times n$ matrix, where $m < 2n^2$, and all coefficients have length $O(|x|^2)$. Now, we can show there exists a polynomial $p(m, n, s)$ of three variables such that any $m \times n$ matrix with rational coefficients all of length less than or equal to s can be reduced in time $p(m, n, s)$. Hence our step (2) can be performed in time roughly $p(2n^2, n, |x|^2)$, which is bounded by a polynomial in $|x|$. □

Note that the index of left nilpotency can be computed by replacing step (1) from Theorem 1 with

(1) Let $C = \{v_i b \mid b \in B\}$.

By an argument identical to that of Theorem 4 we have Theorem 5.

THEOREM 5. *Over either a finite field or Q, left nilpotency can be decided in polynomial time and the index of left nilpotency can be computed in polynomial time.*

Recall that the $O(n^5)$ algorithm of Theorem 2 decided nilpotency, but it did not compute the the index of nilpotency. For some additional running time, we can also compute the index.

THEOREM 6. *Over either a finite field or Q, the index of nilpotency of an n-dimensional algebra can be computed with $O(n^7)$ operations and in polynomial time.*

Proof. We construct a list of sets $B_1, B_2, \cdots, B_{n+1}$ where each B_i is a basis for A^i. The algorithm will successively calculate B_i using the previously calculated B_j's where $j < i$.

Initialize B_1 to the basis for A.
for $i := 2$ to $n+1$
(1) Find all products in $B_1B_{i-1}, B_2B_{i-2}, \cdots, B_{i-1}B_1$
(2) Find a basis B_i for all these vectors.
(3) If $B_i = \{0\}$ then exit.

Here B_jB_k means the finite set of vectors formed by multiplying each member of B_j by a member of B_k. Step (1) involves at most $n-1$ products B_jB_k each calculable in $O(n^5)$ operations, and so it takes $O(n^6)$ operations. Step (2) involves reducing at most n^3 vectors to a basis and can be done in $O(n^5)$ operations. The loop iterates at most n times and so the algorithm needs only $O(n^7)$ operations. Finally, the argument that the algorithm runs in polynomial time, even over Q, is similar to the argument of Theorem 4. □

3. Nillity, left nillity, and a probabilistic approach. In this section we consider algorithms for deciding nillity. Recall that our definition of a nil algebra applied only to power associative algebras. Suppose we are given a power associative algebra, and we wish to decide if it is nil. Note that in an n-dimensional power associative algebra, an element x is nil if and only if $x^{n+1} = 0$. For if x is nil, then it generates an associative nilpotent finite-dimensional subalgebra, and its index of nilpotency is at most $n+1$. It follows that $x^{n+1} = 0$. A remarkable theorem by Dedkov, however, states that if A

is any finite-dimensional power associative algebra over a field of characteristic not equal to 2, 3, or 5, having a nil basis, then A is nil [2]. This leads to an efficient test for nillity: For each basis member x, merely check if $x^{n+1}=0$.

What is the best way to compute x^{n+1}? If x is an arbitrary vector, then it is most efficient to compute a power of x by repeated squarings:

$$x^2, x^{2^2}, \cdots, x^{2^k}, \cdots$$

since only about $\log(n)$ squarings are required, each taking $O(n^3)$ arithmetic operations. However, if x is a *basis* element, then it is slightly faster to compute x^{n+1} using the formula

$$x^{n+1}=x(\cdots x(x(xx))\cdots).$$

Here each multiplication involves a vector with a basis element, an $O(n^2)$ operation. Since there are n such operations, the cost is $O(n^3)$, rather than $\log(n)n^3$. Finally, since there are n basis elements to consider, we have Theorem 7.

THEOREM 7. *In an n-dimensional power associative algebra over any field having characteristic not equal to* 2, 3, 5 *nillity can be tested with $O(n^4)$ arithmetic operations.*

Recall that an *alternative* algebra is one that satisfies the identities

$$(xx)y-x(xy)=0, \tag{5}$$

$$(yx)x-y(xx)=0 \tag{6}$$

for all x and y. These algebras form an important generalization of associative algebras. Although alternative algebras are not in general associative, the subalgebra generated by any two elements is associative [10]. This implies that alternative algebras are power associative, and therefore it is meaningful to speak of alternative nil algebras. If A is an alternative (or, in particular, an associative) finite-dimensional nil algebra, then A is nilpotent [10]. This property, namely that nil finite-dimensional algebras are nilpotent, also holds for many other classes of algebras including Jordan algebras over fields of characteristic not equal to two and others (see [10], [8]).

In associative algebras the concepts of solvable, left nilpotent, and nilpotent, are obviously equivalent. However, as a matter of note, in alternative rings, the concepts of nilpotent and left nilpotent are equivalent, but there exist solvable alternative rings (which cannot be regarded as finite-dimensional algebras) that are not nilpotent [3].

It follows from Theorem 7 that for alternative algebras, Jordan algebras, and the like, after ruling out a few bad characteristics, testing nilpotency takes $O(n^4)$ arithmetic operations, an improvement over the $O(n^5)$ method of Theorem 2.

Unfortunately, when we are presented with an algebra, we do not *know* that it is alternative or power associative, and so we cannot necessarily use Dedkov's result. The property of alternativity can be checked efficiently since it involves only two defining identities of fixed size. But power associativity seems hard to check since it says that for *every* x and for *every* k, *all* associations of k x's are equal.

Let us therefore reformulate the concept of nil so that its definition does not depend on power associativity. For any x, define $x^{[1]}=x$ and for $i\geqq 1$ define $x^{[i+1]}=xx^{[i]}$. We now will call an n-dimensional algebra *left nil* if for each $x\in A$, we have $x^{[n+1]}=0$. It is clear that when A is power associative the notion of left nil coincides with nil.

Assume now that A is any n-dimensional algebra over Q with basis $\{v_i\}$, $i=1,\cdots,n$. We wish to decide if A is left nil. Since A may not be power associative, we cannot rely on Dedkov's result. The problem, therefore, appears hard. In the remainder of this section we will demonstrate an efficient Monte Carlo algorithm.

We first consider the identity

$$x^{[n+1]} = x(\cdots(x(xx))\cdots) = 0. \tag{7}$$

This equation holding for all x is equivalent to A being left nil. Now let $\alpha_1, \cdots, \alpha_n$ be indeterminates and let us write $x = \sum_{i=1}^{n} \alpha_i v_i$ to stand for a generic element in A. We replace x in (7) by $\sum_{i=1}^{n} \alpha_i v_i$, and multiply the expression. Using the structure constants for A, we can simplify this to an expression of the form $\sum_{i=1}^{n} \tau_i v_i$. Here each τ_i is a degree $n+1$ polynomial in the variables $\alpha_1, \cdots, \alpha_n$, with each term having some combination of $n+1$ α's and a coefficient in Q. The algebra A is then left nil if and only if each polynomial τ_i is zero.

Note that the monomials of α's in each τ_i are those we would obtain were we to simplify, using commutativity and associativity, the multivariate polynomial $(\alpha_1 + \cdots + \alpha_n)^{n+1}$. This polynomial has $\binom{2n}{n+1}$ distinct terms. Therefore, explicitly constructing the polynomials τ_i in the manner described above, in order to decide left nillity, is not efficient. Instead, we describe how each τ_i can be shown to be *probably* zero.

LEMMA 2. *Let τ be a polynomial over Q in n variables, $c > 0$, and let I be a subset of Q for which $|I| \geqq c \cdot \deg(\tau)$. Then if τ is not identically zero, the number of elements in I^n which are zeros of τ is at most $c^{-1}|I|^n$.*

Proof. See Corollary 1 of [11, p. 702] for the proof. □

Using the technique of Schwartz [11], we arrive at a probabilistic algorithm for deciding if a polynomial τ is identically zero as follows: First choose I to be any set of elements from Q of cardinality $2 \deg(\tau) = 2(n+1)$. We then select a random n-tuple $y = (y_1, \cdots, y_n)$ from $I \times I \times \cdots \times I$, assign each y_i to α_i, and then evaluate a polynomial τ. This procedure is repeated at most N times. If any of the evaluations produces a nonzero result, then τ is not identically zero. On the other hand, if all evaluations are zero, then by Lemma 2, with $c = 2$, τ is identically zero with probability at least $1 - 2^{-N}$.

In our situation, we really are interested in deciding if all n of the τ_i's are identically zero. Hence we apply the above algorithm to each τ_i. After selecting a random n-tuple $y = (y_1, \cdots, y_n)$ we can evaluate $\tau_i(y_1, \cdots, y_n)$ as follows. Let $x = \sum_{j=1}^{n} y_j v_j$. Now form the sequence

$$x, x^{[2]}, \cdots, x^{[n+1]}.$$

Then $\tau_i(y_1, \cdots, y_n)$ is the coefficient of v_i in $x^{[n+1]}$. This is done up to N times for each τ_i. If any one of the evaluations is nonzero, then (7) fails and the algorithm is terminated. Otherwise, each τ_i is identically zero with probability at least $1 - 2^{-N}$.

Now let $\varepsilon > 0$ be some fixed small number (say 2^{-400}). For a given algebra of dimension n, we choose $N > \log(n/\varepsilon)$. We then test for left nillity in the manner described above. If no nonzero vector is found for the τ_i's then each τ_i is nonzero with probability at most 2^{-N}. The probability, therefore, that at least one of the τ_i's is nonzero is at most $n/2^N$. By choice of N this is less than ε.

Each of the N evaluations takes n multiplications of n-dimensional vectors, where each multiplication takes about $O(n^3)$ operations. Hence for each τ_i, only $O(N \cdot n^4)$ or $O(\log(n) n^4)$ operations are required. This is done for each τ_i, so altogether $O(\log(n) n^5)$ operations are required. Finally, this algorithm runs in polynomial time. The argument is straightforward and uses Lemma 1. Theorem 8 follows.

THEOREM 8. *In an n-dimensional algebra over Q left nillity can be decided probabilistically in polynomial time using $O(\log(n) n^5)$ operations.*

4. An NP-complete problem. In this section we briefly consider the complexity of problems involving algebras described by generators and relations. We assume in this section our algebras are associative. Let $G = \{a_1, \cdots, a_n\}$ be a finite set of generators.

Let W be a set of finite sequences of elements of G. That is, W consists of (associative) words on G. We let ASC (G, W) denote the associative algebra generated by G subject to the relations

(1) For all $w \in W$, $w = 0$.

(2) For each i any product containing two a_i's is zero.

It is clear that ASC (G, W) is nilpotent since the product of any $n+1$ generators must contain two a_i's for some i. Moreover, ASC (G, W) is a finite-dimensional algebra since it is spanned by all words of length $n+1$ or less. Let G^* denote the set of strings over G. Now consider the following decision problem, which we call ASC:

INSTANCE. A finite set G, a finite set of relations $W \subseteq G^*$ as above, and a positive integer k.

QUESTION. Is the index of nilpotency of ASC (G, W) greater than k?

THEOREM 9. ASC is NP-complete.

Proof. It is clear that ASC is in NP since answering yes requires finding a sequence of $k+1$ distinct generators, no subsequence of which is zero by the relations imposed by W. Recall that the problem DIRECTED HAMILTONIAN PATH asks, for a given directed graph, whether there exists a path which visits each vertex exactly once. This problem is NP-complete [5]. We now transform DIRECTED HAMILTONIAN PATH to ASC. Let $D = (V, E)$ be a directed graph with n vertices. We map this to an instance of ASC in which $G = V$, $W = \{a_i a_j \mid (a_i a_j) \text{ is not} \in E\}$, and $k = n$. It is then straightforward to verify that the algebra has index of nilpotency greater than k if and only if the directed graph has a Hamiltonian path. □

5. Unsolvability and local left nilpotency. In recursion theory the *arithmetic hierarchy* is defined as follows. Let Σ_0 be the class of all recursive subsets of natural numbers. For $n \geqq 1$, Σ_n is the class of all sets that are A-recursively enumerable for some $A \in \Sigma_{n-1}$. That is, a set B is a member of Σ_n if there is an oracle program that can enumerate all members of B by making queries of the form "$n \in A$?," for some set $A \in \Sigma_{n-1}$. For each n we also define the class of complements $\Pi_n = \{N - A \mid A \in \Sigma_n\}$. (See [1] or [12].)

Recall that a *reducibility* is a transitive reflexive binary relation. An important example is 1-reducibility, a relation on the class of subsets of the natural numbers. If A and B are sets of natural numbers we say A *is* 1*-reducible to* B (written $A \leqq_1 B$) if there exists a 1-1 recursive (i.e., computable) function f on N such that $n \in A$ if and only if $f(n) \in B$. If Γ is a class of sets we say that B *is* 1*-complete for* Γ if $B \in \Gamma$ and $A \leqq_1 B$ for all $A \in \Gamma$. The purpose of this section is to describe a 1-complete set for the class Π_2 and explain its connection to nilpotence.

For the remainder of this section we assume that F is a fixed field, either finite or countably infinite. If F is infinite we assume that its elements and operations can be described effectively. That is, we assume that there is a 1-1 correspondence that encodes the elements of F with the natural numbers, and there exists an algorithm (on the encoded elements) to compute each field operation. Clearly, a finite extension of Q has this property. From here on, we shall identify a member of F with the number that encodes it.

Next, let g be any 1-1 onto recursive function from N to the set of all finite sequences in F:

$$g(m) = (\alpha_0, \cdots, \alpha_n), \qquad \alpha_i \in F.$$

Also let $\langle x, y \rangle$ be any recursive 1-1 onto map from $N \times N$ to N for which $x, y \leqq \langle x, y \rangle$. (For example, $\langle x, y \rangle = 2^x(2y+1) - 1$ will do.) The maps g and $\langle x, y \rangle$ are thought to be fixed.

Let us now fix a countably infinite set of indeterminates $V=\{v_i\}$, $i=0, 1, \cdots$, to serve as a basis for the vector space A over F of all linear combinations of finitely many v_i's. Let f be a partial recursive function on N. Then f defines a partial function from $V\times V$ to A in the following way. For each i and j for which $f(\langle i,j\rangle)$ is defined we may let

$$v_iv_j=\sum_{k=1}^{n}\alpha_kv_k \tag{8}$$

where $g(f(\langle i,j\rangle))=(\alpha_0,\cdots,\alpha_n)$. This mapping is not necessarily 1-1. For example, if $g(m_1)=(\alpha,\beta)$, $g(m_2)=(\alpha,\beta,0)$, $\langle i_1,j_1\rangle=m_1$, $\langle i_2,j_2\rangle=m_2$, then $v_{i_1}v_{j_1}=v_{1_2}v_{j_2}$. If f is recursive, this defines a multiplication table for an infinite-dimensional algebra with basis V.

We call an algebra with basis V *computable* if it can be obtained from some recursive function f in the above manner. Note that for multiplication to be defined on all pairs of basis elements it is necessary that f be recursive and not just partial recursive. We shall write A_f for this algebra.

Now let ϕ_0, $\phi_1,\cdots$ be a standard numbering of the partial recursive functions on N. (Each ϕ_i is the partial recursive function computed by the ith Turing machine.) We write $\phi_n(j)\uparrow$ to mean $\phi_n(j)$ is undefined, and we write $\phi_n(j)\downarrow$ to mean it is defined. Also, we write $\phi_{n,i}(j)\downarrow$ to mean that the nth Turing machine computes $\phi_n(j)$ in less than or equal to i steps.

An algebra is called *locally* left nilpotent (lln) if every finitely generated subalgebra is left nilpotent. That is, for each subalgebra B generated by a finite set, there exists a k, depending on B, such that $x_1(x_2(\cdots(x_{k-1}x_k)\cdots))=0$ for all $x_i\in B$. We now define

$$\mathrm{LLN}=\{n\mid \phi_n \text{ is recursive and } A_{\phi_n} \text{ is lln}\}.$$

Our goal is to classify LLN in terms of the arithmetic hierarchy.

Let $W(X_1,X_2,\cdots,X_t)$ be a nonassociative word involving t variables, and let $f=\phi_n$ be a partial recursive function. Since f may not be recursive, we must clarify what we mean by the product of v_i's computed by f:

$$W(v_{i_1},v_{i_2},\cdots,v_{i_t}). \tag{9}$$

The word (9) is expanded in the usual way by starting with the innermost pairs of v_i's and applying equation (8). Whenever a product

$$\sum_{i=0}^{n}\alpha_iv_i\cdot\sum_{j=0}^{n}\beta_jv_j$$

must be computed, (8) is only applied to numbers $\langle i,j\rangle$, $\alpha_i\neq 0$ *and* $\beta_j\neq 0$. This procedure defines a partial mapping that we call *the product computed by f.* We now formally define the number of computational steps taken by f to compute the product (9). If $\deg(W)=1$, then f requires zero steps to compute (9). If $\deg(W)=2$ then (9) is of the form $v_{i_1}v_{i_2}$, and the number of steps f requires is the number of steps the nth Turing machine requires to compute $\phi_n(\langle i_1,i_2\rangle)$. For $\deg(W)>2$, we write (9) as

$$R(v_{i_1},v_{i_2},\cdots,v_{i_t})S(v_{i_1},v_{i_2},\cdots,v_{i_t}).$$

Assume $R(v_{i_1},v_{i_2},\cdots,v_{i_t})$ is defined and equal to $\sum_{i=0}^{n}\alpha_iv_i$, and $S(v_{i_1},v_{i_2},\cdots,v_{i_t})$ is defined and equal to $\sum_{j=0}^{n}\beta_jv_j$. Then the number of steps required by f to compute (9) is the sum of the number of steps to compute $R(v_{i_1},v_{i_2},\cdots,v_{i_t})$ plus the number of steps to compute $S(v_{i_1},v_{i_2},\cdots,v_{i_t})$ plus the number of steps to compute v_iv_j for all $\alpha_i\neq 0$ *and* $\beta_j\neq 0$.

Given these formal definitions we now define a predicate $P(n, k, m)$ on $N \times N \times N$ as follows.

> There is some t, $2 \leqq t \leqq m$, such that each right associated product $v_{i_1}(v_{i_2}(\cdots(v_{i_{t-1}}v_{i_t})\cdots))$, where all $i_j \leqq k$, is (1) computed by ϕ_n in at most m steps and (2) equals zero.

Note that since there are only a finite number of such left associated products to check, $P(n, k, m)$ is a recursive predicate.

LEMMA 3. $\mathrm{LLN} = \{n \mid$ *for all* k, *there exists an* m, *such that* $P(n, k, m)\}$.

Proof. Let $n \in \mathrm{LLN}$ and set $f = \phi_n$. Then f is recursive and A_f is locally left nilpotent. Let k be given. Let $B \subseteq A_f$ be the subalgebra generated by $v_0, \cdots, v_k$. By assumption, B is left nilpotent. This implies $B^{[t]} = \{0\}$ for some t. Consider all left associated products having t factors from $\{v_0, \cdots, v_k\}$. Each can be computed by f in a finite number of steps. Define m to be the largest of all such numbers. Then $P(n, k, m)$ is true.

Conversely, assume n is a member of the right side of the equality. We claim $n \in \mathrm{LLN}$. First note that $f = \phi_n$ is recursive: for any k we may find i and j such that $k = \langle i, j\rangle$. By assumption, there exists an m such that $P(k, m, n)$. Hence $v_i(v_i(\cdots(v_iv_j)\cdots))$, since $i, j \leqq k$. This implies $f(k)\downarrow$. Second, we claim A_f is lln. For let $B \subseteq A_f$ be the subalgebra generated by $b_1, \cdots, b_r$. Each b is a linear combination of finitely many v_i's. Let k be the largest such subscript. Let B_1 be generated by $v_0, \cdots, v_k$. Then $B \subseteq B_1$. Since $P(k, m, n)$ for some m, B_1 is lln. Hence B is lln also. □

THEOREM 10. LLN *is* 1*-complete for* Π_2.

Proof. To prove this it suffices to show (1) $\mathrm{LLN} \in \Pi_2$, and (2) $A \leqq_1 \mathrm{LLN}$ for some set A already known to be 1-complete for Π_2. By a well-known characterization of the arithmetic hierarchy (see [1]), a set is in Π_2 if it can be written in the form

$$\{n \mid \text{for all } y, \text{there exists a } z, P(n, y, z)\}$$

where P is a recursive predicate. By Lemma 3, $\mathrm{LLN} \in \Pi_2$. Now let $\mathrm{TOT} = \{n \mid \phi_n$ is recursive$\}$. It is known that TOT is 1-complete for Π_2. We will show $\mathrm{TOT} \leqq_1 \mathrm{LLN}$, which will finish the proof. For each $n \in N$ define the algebra R_n so that for all i, j

$$v_iv_j = \begin{cases} 0 & \text{if } \phi_{n,j}(i)\downarrow, \\ v_{i+j} & \text{otherwise.} \end{cases}$$

That is, v_iv_j is zero if the nth Turing machine halts, within j steps, with input i. Otherwise, v_iv_j is v_{i+j}. Clearly, each R_n is a computable algebra. It is also clear we have a 1-1 recursive map $n \to F(n)$ such that for each $n \in N$, $R_n = A_{\phi_{F(n)}}$. It now suffices to show that $n \in \mathrm{TOT}$ if and only if $F(n) \in \mathrm{LLN}$ (i.e., R_n is locally left nilpotent).

Assume first that $n \in \mathrm{TOT}$ so that ϕ_n is recursive. Let $\{x_i\}$ be a finite set of elements from R_n. We claim this set generates a left nilpotent subalgebra. Each x_i is a finite linear combination of v_j's. Let s be the greatest subscript in all such linear combinations. Then it suffices to show that the subalgebra generated by $\{v_1, \cdots, v_s\}$ is left nilpotent. Since ϕ_n is recursive we may choose t large enough so that

$$\phi_{n,t}(0)\downarrow, \phi_{n,t}(1)\downarrow, \cdots, \phi_{n,t}(s)\downarrow.$$

Now consider any left associated product of $t+1$ elements among $\{v_0, \cdots, v_s\}$, say $v_{i_{t+1}}(v_{i_t}\cdots(v_{i_2}(v_{i_1})\cdots)$. If the right factor is not zero the product becomes $v_{i_{t+1}}v_k$ where $k = \sum_{j=1}^{t} i_j \geqq t$ and $i_{t+1} \leqq s$. Then $\phi_{n,k}(i_{t+1})\downarrow$ and so the final product is zero. This shows R_n is locally left nilpotent.

Conversely, assume n is not a member of TOT (i.e., ϕ_n is not recursive). By assumption on n, there exists an i such that $\phi_{n,j}(i)\uparrow$ for all j. Then any left associated product in R_n of t v_i's is $v_{t\cdot i}$. Hence, R_n is not locally left nilpotent. □

6. Summary and further work. The theme of this paper was to consider one idea from nonassociative algebra, nilpotency, and study it with various computational tools. The ideas from §§ 2 and 3 suggest several questions. For example, although we were able to decide nilpotency in polynomial time, we were not able to decide solvability in polynomial time, at least for finite-dimensional algebras over Q. Can this be done? If not, for what classes of algebras can it be done? Of course, in the case of associative, alternative, Jordan, etc., finite dimensionality and solvability imply nilpotency, and so the problem is solved. But a class of algebras yielding a nontrivial algorithm would be of interest.

In § 3 we noted that in some sense the power of Theorem 7 is wasted unless there is an efficient way to recognize the property of power associativity. It is easy to recognize certain properties that imply power associativity (associativity, alternativity, etc.), but a deeper investigation of power associativity is warranted.

Which of the decision problems in P are also in NC?

The Monte Carlo technique described in § 3 seems powerful enough to handle more general problems. For example, consider the problem in which we are given an algebra A over Q, and an arbitrary nonassociative polynomial f: we wish to decide if f is identically zero in A.

The material in § 5 suggests looking for other unsolvable problems (sets) from nonassociative algebra that are complete for various classes of the arithmetic hierarchy. In particular, it would be nice to identify a problem from nonassociative algebra that is 1-complete for the class of recursively enumerable sets (that is, recursively isomorphic to the halting problem), perhaps something akin to the word problem from group theory.

Finally, a main focus of our work is on the following problem. Let us fix a variety V of nonassociative algebras over a field F, defined by a set of defining identities. For example, V might be the class of alternative algebras over F defined by identities (5) and (6). For each nonassociative polynomial f we wish to decide if f is identically zero for each algebra in V. Assuming that F can be described effectively, this problem is decidable. If the nonassociative polynomials are encoded in a reasonable (i.e., sparse) way, however, there does not *seem* to be any way to solve the problem with a polynomial amount of space. Despite this apparent intractability, much of our work has been to look for better ways to decide if a nonassociative polynomial f is an identity. Here the degree of f is usually small, say at most 10. This problem is quite rich in structure, and offers good opportunity to use many interesting algorithmic and mathematical tools including group representation theory, graph theory, and dynamic programming (see [6], [7]).

REFERENCES

[1] M. DAVIS AND E. WEYUKER, *Computability, Complexity, and Languages*, Academic Press, New York, 1983.

[2] A. I. DEDKOV, *Power-associative algebras with a nil basis*, Algebra i Logika, 24 (1985), pp. 267-277.

[3] G. V. DOROFEYEV, *An example of a solvable but not nilpotent alternative ring*, Uspekhi Mat. Nauk, 15 (1960), pp. 147-150. (In Russian.)

[4] K. FRIEDL AND L. RONYAI, *Polynomial time solutions of some problems in computational algebra*, in Proc. 17th Annual ACM Symposium on Theory of Computing, Providence, RI, 1985.

[5] M. GAREY AND D. JOHNSON, *Computers and Intractability—A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, New York, 1979.

[6] I. R. HENTZEL AND D. J. POKRASS, *A computational and graph theoretic approach to nonassociative algebras*, Cong. Numer., 62 (1988), pp. 241-258.
[7] ———, *Verification of non-identities in algebras*, in Proc. 1988 International Symposium on Symbolic and Algebraic Computation, Springer-Verlag, Berlin, New York, 1989.
[8] D. J. POKRASS AND D. J. RODABAUGH, *On the nilpotency of generalized alternative algebras*, J. Algebra, 49 (1977), pp. 191-205.
[9] L. RONYAI, *Simple algebras are difficult*, in Proc. 19th Annual ACM Symposium on Theory of Computing, New York, 1987.
[10] R. SCHAFER, *An Introduction to Nonassociative Algebras*, Academic Press, New York, 1966.
[11] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701-717.
[12] R. SOARE, *Recursively Enumerable Sets and Degrees*, Springer-Verlag, Berlin, New York, 1987.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 44–70, February 1990

003

THE COMPLEXITY OF VERY SIMPLE BOOLEAN FORMULAS WITH APPLICATIONS*

H. B. HUNT III†‡ AND R. E. STEARNS†§

Abstract. The concepts of **SAT**-hardness and **SAT**-completeness modulo npolylogn time and linear size reducibility, denoted by **SAT**-hard (npolylogn, n) and **SAT**-complete (npolylogn, n), respectively, are introduced. Regardless of whether $\mathbf{P} = \mathbf{NP}$ or $\mathbf{P} \neq \mathbf{NP}$, it is shown that intuitively

Each **SAT**-hard (npolylogn, n) problem requires essentially at least as much deterministic time as,

and

Each **SAT**-complete (npolylogn, n) problem requires essentially the same deterministic time as

the satisfiability problem for 3CNF formulas.

It is proved that the $\leqq$, satisfiability, tautology, unique satisfiability, equivalence, and minimization problems are already **SAT**-complete (npolylogn, n), for very simple Boolean formulas and for very simple systems of Boolean equations. These completeness results are used to characterize the deterministic time complexities of a number of problems for lattices, propositional calculi, combinatorial circuits, finite fields, rings $\mathbf{Z}_k$ $(k \geqq 2)$, binary decision diagrams, and monadic single variable program schemes. A number of these hardness results are "best" possible.

Key words. complexity, **NP**-completeness, **SAT**-completeness, decision problems, Boolean formulas, finite fields, modular arithmetic, binary decision diagrams, program schemes, finite and distributive lattices, fault detection

AMS(MOS) subject classifications. 03G99, 06B99, 06D99, 68Q15, 68M15, 94C10

1. Introduction.

We study the deterministic time complexity of computational problems for very simple Boolean formulas and for very simple systems of Boolean equations. In particular, we study the fundamental problems of $\leqq$, satisfiability, tautology, unique satisfiability, equivalence, and minimization. There are two reasons for this study.

First, the problem instances we consider are so simple that they can be expected to be encountered in any application area. In contrast, a result derived from complex problem instances might be dismissed in some application areas on the grounds that the formula instances used in the hardness proof are not of the form encountered in practice. In general, proofs obtained from simple instances are better evidence of hardness than proofs obtained from general instances.

Second, hardness results for them are more easily extended to other problems. For example, we obtain results for very simple monotone formulas (formulas without **not**) and these results easily generalize to many lattices including all nondegenerate finite lattices.

Although our basic technique is to find reductions from the Satisfiability Problem, we will derive results that are sharper than **NP**-completeness. The disadvantage of

* Received by the editors April 18, 1988; accepted for publication (in revised form) April 4, 1989. A preliminary version of this paper was presented at the Third Annual Symposium on Theoretical Aspects of Computer Science, Paris, Orsay, France. The symposium was supported by the Association Française des Sciences et Techniques de l'Information, de l'Organisation et des Systèmes (AFCET) and the Gesellschaft für Informatik (GI).

† Computer Science Department, State University of New York, Albany, New York 12222.

‡ The research of this author was supported in part by National Science Foundation grants DCR 84-03014 and DCR 86-03184.

§ The research of the author was supported in part by National Science Foundation grant DCR 83-03932.

merely showing **NP**-completeness is that, for all $\varepsilon > 0$, there are **NP**-complete problems that can be solved in time $2^{O(n^{\varepsilon})}$. Even $2^{n^{1/3}}$ algorithms should be considered practical, even though "**NP**-complete" has become associated with "intractable."

Unless explicitly stated otherwise, a *Boolean formula* is a well-formed formula made up of parentheses, variables, and the operators *and*, *or*, and *not*. A *monotone Boolean formula* is a Boolean formula without occurrences of *not*. A *literal* is a variable or a complemented variable. A 3CNF *formula* is the conjunction (*and*s) of *clauses* where each clause is the disjunction (*or*s) of at most three literals. 3DNF *formulas* are defined analogously.

Henceforth, we abbreviate both the satisfiability problem for 3CNF formulas and the set of satisfiable 3CNF formulas by **SAT**. The sharper technique we use here is to use reductions from **SAT** that are npolylogn in time and linear in size (output is linear in input). This leads us to the concepts of **SAT**-hardness (npolylogn, n) and **SAT**-completeness (npolylogn, n) introduced in § 2. In § 2 we see that "**SAT**-complete (npolylogn, n)" means "takes essentially the same deterministic time as the satisfiability problem for 3CNF formulas."

Our key complexity result obtained here concerns the set of formula pairs (F, G) satisfying $F \leqq G$, where F and G are such that

(1) No variable occurs more than once in F or more than once in G,

(2) F is a monotone CNF formula,

(3) G is a disjunction of monotone CNF formulas.

We show that this set of formula pairs has essentially the same deterministic time complexity as **SAT** (i.e., is **SAT**-complete (npolylogn, n)). As corollaries of this basic result, we characterize the deterministic time complexity of a number of basic problems for all finite nondegenerate lattices. Additional applications are presented to logic, circuit analysis and testing, binary decision diagrams, and monadic single variable program schemes. As one corollary, we prove that the recognition of the set of uniquely satisfiable 3CNF formulas requires "essentially the same deterministic time as" **SAT**. This problem has been extensively studied in the literature (see [30]).

A brief outline of this paper follows. In § 2 we introduce the concepts of npolylogn time and linear size reducibility, **SAT**-hardness (npolylogn, n), and **SAT**-completeness (npolylogn, n). We also show that two important reduction procedures can be performed on npolylogn time and linear size bounded Turing machines. In § 3 we present our main deterministic time complexity results for the $\leqq$, satisfiability, tautology, unique satisfiability, equivalence, and minimization problems for very simple Boolean equations and for very simple systems of Boolean equations. Theorem 3.3 and Corollary 3.4 are of special importance to the remainder of the paper. In § 4 we use the results and techniques of §§ 2 and 3 to characterize the deterministic time complexities of a number of basic problems (see Fig. 1 in § 4.1) for each nondegenerate finite lattice. Additional applications are presented to logic and to circuit analysis and testing. In § 5 we use the results and techniques of §§ 2 and 3 to characterize the deterministic time complexities of a number of basic problems for each finite field, each ring $\mathbf{Z}_k$ $(k \geqq 2)$, binary decision diagrams, and monadic program schemes.

The remainder of this section consists of definitions, notation, and basic results about complexity theory, lattices, and Boolean algebras used in this paper. We assume that the reader is familiar with the complexity classes **P**, **NP**, and **coNP**, polynomial reduciblity, **NP**-hardness and **NP**-completeness, and **coNP**-hardness and **NP**-completeness; otherwise, see [18]. We denote the set of natural numbers by **N**. Throughout this paper by "Turing machine," we mean "multiple-tape Turing machine."

The following problems for Boolean formulas are considered:

(1) The $\leqq$*problem*, i.e., the problem of determining, for Boolean formulas F and G, if $F \leqq G$, i.e., if G equals 1 whenever F equals 1.

(2) The *satisfiability problem*, i.e., the problem of determining if a Boolean formula F is satisfiable.

(3) The *tautology problem*, i.e., the problem of determining if a Boolean formula F is a tautology.

(4) The *unique satisfiability problem*, i.e., the problem of determining, for a Boolean formula F, if there exists exactly one assignment $\mathbf{v}$ of values from $\{0, 1\}$ to the variables of F such that F takes on the value 1 under $\mathbf{v}$.

(5) The *equivalence problem*, i.e., the problem of determining, for Boolean formulas F and G, if F and G denote the same function.

(6) The *minimization problem*, i.e., the problem of finding, given a Boolean formula F, an equivalent Boolean formula G such that the number of occurrences of symbols in G is minimal.

THEOREM 1.1 [15], [18]. *The set of tautological* 3DNF *formulas is* **coNP**-*complete*; *and the set of satisfiable* 3CNF *formulas is* **NP**-*complete.*

DEFINITION 1.2. An *algebraic structure* $\mathbf{S}$ is a nonempty set S, called the *domain* of the structure, together with a nonempty set of operations of various arities on S. $\mathbf{S}$ is said to be *nondegenerate* if $|S| \geqq 2$. $\mathbf{S}$ is said to be *finite* if $|S| < \infty$, and in addition if $\mathbf{S}$ has only finitely many operations, each of finite arity. □

DEFINITION 1.3. A *lattice* $\mathbf{S} = (S, \vee, \wedge)$ is an algebraic structure with domain S such that $\vee$ and $\wedge$ are commutative, associative, and idempotent binary operations on S such that, for all $x, y \in S$, $x \vee (x \wedge y) = x \wedge (x \vee y) = x$. A *distributive lattice* $\mathbf{S} = (S, \vee, \wedge)$ is a lattice such that, for all $x, y, z \in S$, $x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z)$ and $x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z)$. A lattice $\mathbf{S} = (S, \vee, \wedge)$ is *finite* if $|\mathbf{S}| < \infty$.

Let $\mathbf{S} = (S, \vee, \wedge)$ be a lattice. Let $\leqq$ be the partial order on S defined by $x \leqq y$ if and only if $x \vee y = y$. An element a of S such that $a \leqq b$ for all $b \in S$ is said to be the *minimal element* on $\mathbf{S}$ and is denoted by 0. An element a of S such that $b \leqq a$ for all $b \in S$ is said to be the *maximal element* of $\mathbf{S}$ and is denoted by 1. Let $\mathbf{S} = (S, \vee, \wedge)$ be a lattice with minimal element 0. An element b of S such that $0 < b$ on $\mathbf{S}$ but there exists no $c \in S$ for which $0 < c < b$ on $\mathbf{S}$ is said to be an *atom* of $\mathbf{s}$. A lattice $\mathbf{S} = (S, \vee, \wedge)$ is said to be a *finite depth lattice* if there exists $k \in \mathbf{N}$ such that

$$x_l < \cdots < x_2 < x_1 \quad \text{on } \mathbf{S} \text{ implies } l \leqq k. \qquad \square$$

A *Boolean Algebra* has operators $\wedge$, $\vee$, and $\sim$ and constants **0** and **1** where $\wedge$, $\vee$, and $\sim$, behave as set intersection, union, and complement, respectively, **0** behaves as the empty set, and **1** behaves like the universal set. Formal axioms can be found in [1], [7], and [43]. We let **BOOLE** be the two-element Boolean algebra of everyday logic. We let **BIN** be the two-element distributive lattice, namely, **BOOLE** without the negation (or complement) operator.

THEOREM 1.4 [7]. (1) *Let* $\mathbf{L} = (S, \vee, \wedge)$ *be a nondegenerate distributive lattice. Let* F *and* G *be formulas on* $\mathbf{L}$ *involving only variables, parentheses,* $\vee$, *and* $\wedge$. *Then,* $F \leqq G$ *on* $\mathbf{L}$ *if and only if* $F \leqq G$ *on* **BIN**; *and* $F = G$ *on* $\mathbf{L}$ *if and only if* $F = G$ *on* **BIN**.

(2) *Let* $\mathbf{L} = (S, \vee, \wedge, \sim, 0, 1)$ *be a nondegenerate Boolean algebra. Let* F *and* G *be formulas on* $\mathbf{L}$ *involving only variables, parentheses,* $\vee$, $\wedge$, $\sim$, 0, *and* 1. *Then,* $F \leqq G$ *on* $\mathbf{L}$ *if and only if* $F \leqq G$ *on* **BOOLE**; *and* $F = G$ *on* $\mathbf{L}$ *if and only if* $F = G$ *on* **BOOLE**.

The importance of Theorem 1.4 is that hardness results concerning formulas on **BIN** immediately generalize to arbitrary nondegenerate distributive lattices and results about **BOOLE** immediately generalize to arbitrary Boolean Algebras.

In general, formulas over an algebraic structure on domain S will involve variables, operators, and some notation for elements in S. We generally focus on "constant-free"

formulas involving only variables and operators. The distinction between operators and constants can be blurred by the presence of zero-ary operators (such as **0** and **1** in Boolean algebras). We call formulas with these zero-ary operators "constant-free" since they can be interpreted as formulas independent of the domain.

Restricting ourselves to constant-free formulas does not weaken hardness results since we certainly expect them to be included among formulas encountered in practice. We seek results on constant-free formulas that apply to the class of all algebras with the specified operators. Classes of formulas with domain specific constants can sometimes be harder than constant-free formulas due to the complexity of manipulating constants. The complexity of manipulating constants (i.e., the complexity of arithmetic) is not a topic of this paper.

In the case of finite algebraic structures **S**, the domain of the structure can be specified by giving distinct names to its elements. The complexity of arithmetic on such a structure **S** is not an issue, since **S**'s operators can be specified by tables and have constant cost.

DEFINITION 1.5. Let **S** be an algebraic structure with domain S. By a *representation* of **S**, we mean a set of $|S|$ distinct constant symbols denoting the elements of S. □

The algebraic structures we use here are Boolean algebras, lattices, logics, and rings, which have standard infix notation for formulas. In general, the results apply to any of the easily parsed formula notations. By the *size* of a formula F denoted by $\|F\|$, we mean the number of occurrences of symbols in F, where each occurrence of a variable, operator, constant, or parenthesis is treated as a single occurrence. For example, $\|(x_{135} \text{ or } x_{321})\| = 5$. The *size* of an equation or a system of equations is defined analogously. This is the natural measure since variables and constants are the objects on which reductions are defined. When considering the time of a reduction on a Turing machine, however, we will take into account the fact that the infinite variable set must actually be represented by strings on some finite alphabet.

We like to measure time complexity as a function of input size rather than input length. When doing this, we use the symbol $\|w\|$ instead of the traditional n. Thus we use $L \in \text{DTIME}(F(\|w\|))$ to mean the time required to test string w for membership in L is $F(\|w\|)$ or fewer Turing machine operations. It is assumed that a reasonably efficient encoding of variables into strings is used when a formula is presented to a Turing machine. Specifically, we assume the length of the Turing machine input is at worst $O(\|w\| \log \|w\|)$.

Let F be a formula on an algebraic structure **S** with domain S. Let **v** be an assignment of values from S to the variables of F. We denote the value taken on by F under **v** by $\mathbf{v}[F]$.

2. Preliminary results. Here we present our hardness concepts and prove their implications for complexity. The objective is to establish stronger relationships than **NP**-hardness. We close the section with efficient time and size bounded Turing machine algorithms for two basic transformations that serve as subroutines in later sections.

In what follows, let Σ and Δ be finite nonempty alphabets; and let L and M be languages over Σ and over Δ, respectively.

DEFINITION 2.1. We say that L is npolylogn *time and linear size reducible* to M if there exists an integer $k \geqq 1$ and a function $f: \Sigma^* \to \Delta^*$ computable by an $O(n(\log n)^k)$ time-bounded deterministic multiple tape Turing machine such that:

(i) For all $x \in \Sigma^*$, $x \in L$ if and only if $f(x) \in M$; and

(ii) There exists $c > 0$ such that, for all $x \in \Sigma^*$, $\|f(x)\| \leqq c \cdot \|x\|$. □

DEFINITION 2.2. We say that L is **SAT**-hard (npolylogn, n), read "L is **SAT**-hard modulo npolylogn time and linear size reducibility," if **SAT** is npolylogn time and

linear size reducible to L (in which case L is also **NP**-hard) or un**SAT** (the set of unsatisfiable 3CNF formulas) is npolylogn time and linear size reducible to L (in which case L is **coNP**-hard). We say that L is **SAT**-complete (npolylogn, n), read "L is sat complete modulo npolylogn time and linear size reducibility," if L is **SAT**-hard (npolylogn, n), and L is npolylogn time and linear size reducible to either **SAT** or un**SAT**. □

We will use the term Turing-**SAT**-complete (npolylogn, n) if the above conditions hold for Turing reductions instead of many-one reductions, where npolylogn is the time spent to create a bounded number of problem instances of linear size.

The basic deterministic time hardness properties of **SAT**-hard and **SAT**-complete (npolylogn, n) languages are summarized in Proposition 2.3. Part (1) of the proposition applies if $\mathbf{P} = \mathbf{NP}$ and says that, in the case of **SAT**-completeness, the complexity of L and **SAT** are bounded by the same polynomials. Part (2) applies if the **NP**-complete problems take exponential time and says, that in the case of **SAT**-completeness, the complexities of L and **SAT** have similar polynomials in their exponents. Thus, intuitively, this proposition says that:

(1) If L is **SAT**-hard (npolylogn, n), then L requires "essentially at least as much deterministic time as **SAT**," and

(2) If L is **SAT**-complete (npolylogn, n), then L requires "essentially the same deterministic time as **SAT**."

We note that each **SAT**-hard (npolylogn, n) language is either **NP**- or **coNP**-hard, and that each **SAT**-complete (npolylogn, n) language is either **NP**- or **coNP**-complete.

PROPOSITION 2.3. *Let L be a language and let $T(n)$ be any increasing function such that, for all k, $T(n) \geqq n(\log n)^k$ for almost all n:*

(1) *If L is* **SAT**-hard (npolylogn, n) *and* $L \in \text{Dtime}(T(\|w\|))$, *then* $\mathbf{SAT} \in \text{Dtime}(T(c\|w\|))$ *for some constant c.*

(2) *If L is* **SAT**-complete (npolylogn, n), *then* $L \in \text{Dtime}(T(c_1\|w\|))$ *for some c_1 if and only if* $\mathbf{SAT} \in \text{Dtime}(T(c_2\|w\|))$ *for some c_2.*

(Statement 2 also applies to Turing-**SAT**-complete (npolylogn, n) problems.)

Proof. If L can be done in time $T(\|w\|)$, then the reduction permits **SAT** to be done in time $O(\text{npolylogn}) + T(O(n))$, which is $O(T(c\|w\|))$ for some constant c. This proves Part (1). If L is complete, there is a corresponding reduction back to **SAT** and Part (2) is proved. These remarks apply equally well to Turing reductions. □

If, as we suspect, **SAT** requires deterministic time $2^{O(n^k)}$ for some k, the **SAT**-hard(npolylogn, n) problems will also take at least $2^{O(n^k)}$ time. Complete problems will have the property that $L \in \text{Dtime}(2^{O(\|w\|^k)})$ if and only if $\mathbf{SAT} \in \text{Dtime}(2^{O(\|w\|^k)})$. If $\mathbf{P} = \mathbf{NP}$ and **SAT** only requires time $O(n^k)$, then the hard problems will also require time $O(n^k)$ and complete problems will have the property that $L \in \text{Dtime}(O(\|w\|^k))$ if and only if $\mathbf{SAT} \in \text{Dtime}(O(\|w\|^k))$.

The next proposition is used extensively in this paper.

PROPOSITION 2.4. *Let Θ be any nonempty finite set of Boolean operators. Then there exists a constant $c > 0$ and a deterministic $O(\text{npolylogn})$ time-bounded Turing machine T such that, when given a system of Boolean equations* **S** *involving operators from Θ and the constants* 0 *and* 1 *as input, T outputs a* 3CNF *formula $F_{\mathbf{S}}$ such that*

(1) $\|F_{\mathbf{S}}\| \leqq c \cdot \|\mathbf{S}\|$, *and*

(2) *The number of satisfying assignments of $F_{\mathbf{S}}$ equals the number of satisfying assignments of* **S**.

Proof. This reduction can be done by standard techniques using the principles from [6] and standard compiling techniques. We outline the reduction as a sequence of steps with the expectation that the reader can verify that each step can be carried

out by a Turing machine in the required time and satisfying conditions (1) and (2) of the proposition. In practice, the ideas can be fit into a one-pass algorithm.

Step 1. The system S can be thought of as a list of formula pairs where the two formulas in each pair are to be made equal. Replace each operator occurrence Θ in the input formulas with a pair (Θ, v), where v is a variable distinct from the input variables and the other new variables associated with other operator occurrences.

Step 2. Translate the string into a sequence of equations where the left-hand formula has no operators and the right-hand formula has at most one operator. For each pair (Θ, v) in the input to this step, there will be an equation $v = \Theta(x_1 \cdots x_k)$ where Θ is k-ary and $x_1 \cdots x_k$ are variables or constants representing the operands associated with the occurrence of Θ in the input. For each formula pair of S, the output will have equation $x = y$ where x and y are the variables (or constants) representing the two formulas.

Step 3. For each equation, there is a 3CNF formula that expresses the same Boolean relationship as the equation. The output of the procedure is the conjunction of all these formulas.

Because Θ is a finite set of Boolean operators, we are dealing with a finite set of transformations of individual equations into 3CNF. Therefore this last step is linear size bounded. □

The next proposition asserts the existence of a subroutine for marking variable occurrence quickly on a Turing machine.

PROPOSITION 2.5. *There is a deterministic* npolylogn *time-bound Turing machine that, given a sequence F of symbols and variables as input, replaces each variable x by a pair* (x, k) *where k is the integer such that* (x, k) *is the replacement for the kth occurrence of x in F.*

Proof. The set of integers $\{1, 2, \cdots, n\}$, when denoted by their binary numerals, can be sorted deterministically in npolylogn time on a Turing machine using a standard merge-sort algorithm. The Turing machine of the statement of the proposition uses such an npolylogn time sorting algorithm as a subroutine. Let k be the number of variable occurrences in F and let the ith occurrence be x_{j_i}. This machine, given F as input, executes the following five steps:

Step 1. Extract the string $F_1 = (x_{j_1}, 1) \cdots (x_{j_k}, k)$ from F.

Step 2. Sort the pairs in F_1 according to the index of these variables (and preserving the original order among occurrences of the same variable). Call the result F_2.

Step 3. Make each pair (x, i) of F_2 into a triple (x, i, l) where (x, i) is the lth occurrence of $x$in F_2. This can be done in npolylogn time because Step 2 has made the occurrences of x adjacent. Call the result F_3.

Step 4. Sort the triples in F_3 according to the second component. This restores the variable occurrence to the original order of F. Call this result F_4.

Step 5. Take the triples from F_4 and attach the third component to the corresponding occurrence in F. This is the desired output. □

We note that, after executing Step 2 of the algorithm immediately above, the Turing machine of the proof can be modified to output in npolylogn time and in order of increasing variable subscript both the variables of F and the numbers of times they occur in F.

3. Hard problems for very simple formulas and systems of equations. We study the deterministic time complexities of the $\leqq$, satisfiability, unique satisfiability, tautology, equivalence, and minimization problems for Boolean formulas and systems of Boolean equations. More specifically, we describe very simple formulas and systems for which

these problems are hard. In each case, the results are on the boundary of **NP** in that the obvious further simplifications result in problems in **P**. Some of the results, most notably Theorem 3.3, say that two "easy problems" can be combined in simple ways to get problems that are "hard as they can be."

In the first theorem, the satisfiability problem for 3CNF formulas with $\leqq 3$ repetitions per variable is considered. The **NP**-hardness of this problem is known and is mentioned in [18]. To put this hard problem into the framework of **SAT**-completeness (npolylogn, n), we must show that reductions exist with the required time and size bound. No reduction is cited in [18].

THEOREM 3.1. *The Satisfiability Problem is* **SAT**-complete (npolylogn, n) *for* CNF *formulas with* $\leqq 3$ *literals per clause and* $\leqq 3$ *repetitions per variable. The Tautology Problem is* **SAT**-complete (npolylogn, n) *for* DNF *formulas with* $\leqq 3$ *literals per term and* $\leqq 3$ *repetititions per variable.*

Proof. To verify that these two problems are **SAT**-hard (npolylogn, n), it suffices by duality to give an npolylogn time and linear size reduction from the 3SAT to 3SAT for CNF formulas with $\leqq 3$ repetitions per variable. The following reduction can be used to reduce any Boolean formula f to a Boolean formula f_2 such that

(a) No variable occurs more than three times in f_2, and

(b) f is in SAT if and only if f_2 is in SAT.

Let $x_1, \cdots, x_n$ be the variables occurring more than one time in f. Let $i_1, \cdots, i_n$ be the number of occurrences of $x_1, \cdots, x_n$, respectively, in f. For $1 \leqq j \leqq n$ and $1 \leqq k \leqq i_j$, let the variables $x_{j,k}$ be distinct variables. Let f_1, F, and f_2 be the Boolean formulas defined as follows:

(i) f_1 is the CNF Boolean formula that results from f by replacing, for $1 \leqq j \leqq n$ and $1 \leqq k \leqq i_j$, the kth occurrence to the variable x_j in f by the variable $x_{j,k}$. Variables appear in f_1 only once.

(ii) For $1 \leqq j \leqq n$, let $g_j = t_{j,1}$ *and* $\cdots$ *and* t_{j,i_j} where $t_{j,k} = (x_{j,k}$ *or* (not $x_{j,k+1}))$ for $k < i_j$ and $t_{j,i_j} = (X_{J,ij}$ or (not $x_{j,1}))$. Formula g_j is true if and only if each $t_{j,k}$ is true, which can happen if and only if all the variables with first subscript j have the same value. These variables appears appear in g_j only twice.

(iii) Let F be the CNF formula g_1 *and* g_2 *and* $\cdots$ *and* g_n. Each variable appears in F two times.

(iv) f_2 is the Boolean formula (F *and* f_1).

The formula F is true if and only if, for all assignments $\mathbf{v}$ of values from $\{0, 1\}$, $\mathbf{v}[x_{i,j}] = \mathbf{v}[x_{i,k}]$ for all i, j, k. Thus, it is easily seen that the formula f_2 satisfies the assertions a and b immediately above. Clearly, $\|f_2\| = \|F\| + \|f_1\| + 3 = O(\|f\|)$. Also clearly when f is a CNF formula, so is f_2. Finally, by using the deterministic npolylogn time-bounded Turing machine of Proposition 2.5 as a subroutine, it is easy to see that the reduction can be carried out on a deterministic npolylogn time-bounded Turing machine. □

We note that the reduction of the proof of Theorem 3.1 is parsimonious, i.e., preserves the number of satisfying assignments.

There are two obvious ways the satisfiability problem of Theorem 3.1 can be simplified. One is to allow only two literals per clause and the other is to restrict variables to at most two occurrences. By the results of Cook [15] and Tovey [51] respectively, both these problems are in **P**. The next result shows that we can get hard problems with only two repetitions if we consider formulas more complex than CNF. However, we do not need to go beyond the conjunction of DNF formulas to get problems that are as hard as they can be.

THEOREM 3.2. *Consider the set of Boolean formulas f such that*

(i) *f is a conjunction of* DNF *formulas*:

(ii) *Each variable of f occurs exactly once complemented and once uncomplemented. The satisfiability problem for formulas in this set is* **SAT**-complete (npolylogn, n) *and* **NP**-hard.

Proof. Let f be a CNF formula with $\leqq 3$ literals per clause and $\leqq 3$ repetitions per variable. Formula f can be reduced to a simpler problem if some variable appears only uncomplemented. Just replace the variable by the constant 1 and simplify. A similar simplification can be done if a variable appears only complemented. Therefore, without loss of generality, we may further assume the following:

(1) Each variable of f appears both complemented and uncomplemented.

(2) No variable of f occurs twice complemented (by replacing a variable by its complement a varaible that appears twice complemented and hence once uncomplemented can be converted into a once complemented variable).

Let $x_1, \cdots, x_k$ be the variables of f that occur three times in f. For $1 \leqq j \leqq k$, let y_{j1} and y_{j2} be distinct variables. Let f' be the Boolean formula that results from f by replacing, for $1 \leqq j \leqq k$,

> The first uncomplemented occurrence of x_j in f by y_{j1}, the second uncomplemented occurrence of x_j in f by y_{j2}, and the occurrence of $\bar{x}_j$ in f by $\bar{y}_{j1}$ *and* $\bar{y}_{j2}$.

Under this transformation, the clauses of f become DNF formulas and f' is the conjunction of DNF formulas. Thus condition (i) is satisfied and it is easy to see that (ii) is also satisfied. Also, clearly $\|f'\| = O(\|f\|)$. We claim the following:

(3.2.1) f is satisfiable if and only if f' is satisfiable.

(3.2.2) f' is constructible from f on a deterministic npolylogn time-bounded Turing machine.

The correctness of claims (3.2.1) and (3.2.2) implies the theorem.

It is obvious that a satisfying assignment for f can be made into a satisfying assignment for f'. Therefore, to prove the correctness of claim (3.2.1), it suffices to show the following:

> If there exists an assignment v of values from $\{0, 1\}$ to the variables of f' such that $v[f'] = 1$ and such that $v[y_{j1}] \neq v[y_{j2}]$ for some j with $1 \leqq j \leqq k$, then there exists an assignment w of values from $\{0, 1\}$ to the variables of f' such that $w[f'] = 1$ and, for $1 \leqq j \leqq k$, $w[y_{j1}] = w[y_{j2}]$.

For each such assignment v, let w be the assignment that is the same as v accept that, for $1 \leqq j \leqq k$, if $v[y_{j1}] \neq v[y_{j2}]$, then $w[y_{j1}] = w[y_{j2}] = 1$. Since f' is a Boolean formula monotone in literals, $1 = v[f'] \leqq w[f']$. Finally, the correctness of claim (3.2.2) follows from the proof of Proposition 2.5 (using literals instead of variables). □

We might imagine, intuitively, that the "hard" problem instances must be constructed in a series of steps, each of which combines problems that are slightly less hard. Our next result shows that such intuition is wrong, and we can construct problems that are as hard as they can be by combining two "easy problems" with a single binary operator. In this case, the easy problems are monotone Boolean formulas that are the disjunctions of CNF formulas and that do not have variables occurring more than once. These are "easy problems" in that they are always satisfiable, are never tautologies, and their solutions can be counted quickly.

THEOREM 3.3. *Let F and G be Boolean formulas such that*

(i) *No variable occurs more than once in F or more than once in G,*

(ii) *F is a monotone* CNF *formula,*

(iii) *G is the disjunction of monotone* CNF *formulas.*

Then, the following problems are **SAT**-complete (npolylogn, n):

(1) *Determining if* $F \leqq G$,

(2) *Determining if the formula* (F *and* $(\sim G)$) *is satisfiable,*

(3) *Determining if the formula* (G *or* $(\sim F)$) *is a tautology,*

(4) *Determining if the formula* $(F \Rightarrow G)$ *is a tautology, and*

(5) *Determining if the system of equations* $F = 1$ *and* $G = 0$ *has a solution.*

Problems (2) *and* (5) *are also* **NP**-*hard and Problems* (1), (3), *and* (4) *are* **coNP**-*hard. The problems remain hard if*

(ii′) *F is the conjunction of monotone* DNF *formulas, and*

(iii′) *G is a monotone* DNF *formula.*

Proof. For all Boolean formulas F and G, the following are obviously equivalent:

(1) $F \leqq G$,

(2) The formula (F *and* $(\sim G)$) is not satisfiable,

(3) The formula (G *or* $(\sim F)$) is a tautology,

(4) The formula $(F \Rightarrow G)$ is a tautology, and

(5) The system $F = 1$ and $G = 0$ has no solution.

Thus to prove the theorem, it suffices to prove that the problem of (1) is **SAT**-complete (npolylogn, n) for monotone Boolean formulas F and G satisfying conditions (i)-(iii) of the theorem.

Proof of (1). To prove **SAT**-hardness (npolylogn, n) and **coNP**-hardness, we give an npolylogn time and linear size reduction that maps a formula f monotone in literals to an inequality ($\leqq$) of monotone Boolean formulas such that f is a tautology if and only if the output inequality holds. When applied to formulas that are the disjunction of CNF formulas where each variable appears exactly once complemented and exactly once uncomplemented, the procedure will output F and G satisfying conditions (i)-(iii). Thus by the dual of Theorem 3.2 and the transitivity of npolylogn time and linear size reducibility, we can conclude that the problem of (1) is **SAT**-hard (npolylogn, n) for formula F and G satisfying conditions (i)-(iii) of the theorem. Given this, the **SAT**-completeness (npolylogn, n) of the problem follows immediately from Proposition 2.4.

Let f be a Boolean formula monotone in literals. Let $x_1, \cdots, x_n$ be variables occurring in f. Let $y_1, \cdots, y_n$ be distinct variables other than $x_1, \cdots, x_n$. Let f' be the monotone Boolean formula that results from f by replacing, for $1 \leqq i \leqq n$, the occurrences of (*not* x_i) in f by y_i. Let F_n be the monotone Boolean formula (x_1 *or* y_1) *and* $\cdots$ *and* (x_n *or* y_n). Clearly, the formulas f' and F_n are constructible from f determinstically in linear time. Clearly f' and F are monotone. We claim that

(3.3.1) f is a tautology if and only if $F_n \leqq f'$.

To prove (3.3.1), assume v is an assignment of values from $\{0, 1\}$ to variables $x_1, \cdots, x_n$ such that $v[f] = 0$. Consider the assignment w to $x_1, y_1, \cdots, x_n, y_n$, such that $w[x_i] = v[x_i]$ and $w[y_i] =$ *not* $v[x_i]$ for all $i \leqq n$. Clearly, $w[f'] = 0$ since f and f' are identical formulas after their respective substitution of values for variables and literals. Also $w[F_n] = 1$ because $w[x_i$ *or* $y_i] = 1$ for all $i \leqq n$ by construction. Therefore $w[F_n] > w[f']$ and $F_n \leqq f'$ fails.

To prove (3.3.1) in the reverse direction, consider an assignment w of values from $\{0, 1\}$ to $x_1, y_1, \cdots, x_n, y_n$ such that $w[F_n] = 1$ and $w[f'] = 0$. Consider the assignment v from $\{0, 1\}$ to $x_1, \cdots, x_n$ such that $v[x_i] = w[x_i]$ for all i. A key fact is that this assignment also satisfies $v[\bar{x}_i] \leqq w[y_i]$. This fact follows from $w[x_i$ *or* $y_i] = 1$ (because

$w[F_n]=1$) and $v[x_i]=w[x_i]$ (by construction). Again consider the formulas f and f' after substitution for variables and literals. The resulting expressions are identical except that certain occurrences of 1 in f' may be 0 in f. (The reverse situation is prevented by the "key fact.") Because the expressions are monotone, $w[f'] \geqq v[f]$. But since $w[f']=0$, $v[f]=0$ and f is not a tautology. Thus (3.3.1) is proved.

Finally, we must verify that no variable is repeated in f' or F_n, when f has variables appearing once complemented and once uncomplemented. But clearly f' has exactly one occurrence of each x_i and exactly one occurrence of each y_i. F_n is constructed to have only single occurrences independently of f.

The statements about **NP**-hardness and **coNP**-hardness are evident from the proof. To prove the result for alternative conditions (ii′) and (iii′), observe that $F \leqq G$ implies $\sim G \leqq \sim F$. Applying DeMorgan's laws and replacing variables by their complements then gives result (1) and the others follow as above. □

Although parts (1)-(5) of Theorem 3.3 are really five ways of saying the same thing, they have different applications. Part (1) is a statement involving only $\wedge$, $\vee$, and $\geqq$ and it can thus be viewed as a statement about distributive lattices. Part (4) can be viewed as a statement about logics without a negation operator. Parts (2) and (3) say Boolean formulas become hard the very first time tractable formulas are combined. Part (5) addresses systems of equations in which the constants 0 and 1 are available.

The "cause" of hardness in Theorem 3.3 is the two levels of *or* in G allowed by condition (iii) or the two levels of *and* in G allowed by condition (ii′). If G has only one level of *or*s and F only one level of *and*s, $F \leqq G$ becomes easy, even under the following circumstances:

(a) F is the disjunction of CNF formulas, not necessarily monotone, in which each CNF formula has no repeated variables.

(b) G is the conjunction of DNF formulas, not necessarily monotone, in which each DNF formula has no repeated variables.

To see this, observe $\vee \mathrm{CNF}_i \leqq \wedge DNF_j$ if and only if $\mathrm{CNF}_i \leqq \mathrm{DNF}_j$ for all i and j if and only if $(\neg \mathrm{CNF}_i \vee \mathrm{DNF}_j)$ is a tautology for all i and j. Under DeMorgan's laws, $(\neg \mathrm{CNF}_i \vee \mathrm{DNF}_j)$ becomes a DNF formula in which no variable occurs more than twice, and the tautology problem for such formula is known to be in **P**.

The following corollary shows that the equivalence of monotone formulas is also hard in simple cases:

COROLLARY 3.4. *Testing $f = g$ for formula is* **coNP**-*hard and* **SAT**-complete (npolylogn, n) *even if*

(1) *f is a monotone* CNF *formula, and*

(2) *g is the disjunction of monotone* CNF *formulas.*

Proof. The proof follows from part (1) of Theorem 3.3, since $F \leqq G$ if and only if $F = F \wedge G$. □

The next result extends Theorem 3.3 to questions about unique satisfiability:

THEOREM 3.5. *The following problems are* **NP**-*hard and* **SAT**-hard (npolylogn, n):

(1) *Determine if a system of two monotone Boolean equations has a unique solution, even if no variable occurs more than three times.*

(2) *Determine if a* 3CNF *formula has a unique solution, even if no variable occurs more than three times.*

Proof. We first show problem (1). Let F and G be monotone Boolean formulas such that no variable occurs more than once in F and more than once in G. Let $x_1, \cdots, x_n$ be the variables occuring in F or in G. Let $y_1, \cdots, y_n$ be n additional

variables. Then, the following are equivalent:

(i) The system of equations

$$F=1, \qquad G=0$$

does not have a solution, and

(ii) The system of equations

$$(x_1 \wedge y_1) \vee \cdots (x_n \wedge y_n) \vee G=0, \qquad F \vee (y_1 \wedge \cdots \wedge y_n)=1$$

has a unique solution.

To see the equivalence note that $x_1=\cdots=x_n=0$ and $y_1=\cdots=y_n=1$ is a solution of the two equations of (ii). Any other solution of the equations of (ii) is a solution of the equations of (i); and any solution of the equations of (i) can be extended to an additional solution of the equations of (ii) by setting $y_1=\cdots=y_n=0$. Since (ii) can be obtained from (i) in npolylogn time and (i) is **NP**-hard and **SAT**-hard (npolylogn, n) by part (5) of Theorem 3.3, we have part (1) of this theorem. Problem (1) is reduced to Problem (2) by the procedure of Proposition 2.4. (It is easily verified that this procedure preserves the "at most three repetitions" property.) □

We next show that the unique satisfiability problems of the previous theorem have "essentially the same hardness" as **SAT**. In this case we will be using a Turing reduction instead of a many-one reduction so we have a result for Turing-completeness instead of completeness. Actually the reduction is a simple norm 2 truth-table reduction.

PROPOSITION 3.6. *The problems of Theorem* 3.5 *are* Turing-**SAT**-complete (npolylogn, n).

Proof. We need only consider the unique 3CNF problem (problem (2) of Theorem 3.5) since the first problem has already been efficiently reduced to the second in the proof of Theorem 3.5.

Let f be a 3CNF formula. Let $x_1, \cdots, x_n$ be the variables occurring in f. Let $y_1, \cdots, y_n$ be additional variables. Then, f is uniquely satisfiable if and only if

f is satisfiable, and the Boolean formula $f(x_1, \cdots, x_n) \wedge f(y_1, \cdots, y_n) \wedge (x_1 \oplus y_1 \vee \cdots \vee x_n \oplus y_n)$ is not satisfiable.

Thus unique satisfiability can be solved by solving satisfiability twice. Each of these formulas is linear in the size of the original. □

It is already known that unique **SAT** is **coNP**-hard and can be solved in polynomial time using **NP** twice as an oracle (see [24], [8]). Our proofs imitate some of the past techniques, verifying the time and size of the reductions and applying them to the special case of limited variable occurrences.

Consider the class of Boolean formulas where no variable appears more than twice. We have a polynomial time algorithm that decides whether such a formula has a unique solution. (We provide this algorithm in the Appendix.) Theorem 3.2 thus tells us that this is a class of formulas where satisfiability is **NP**-complete and unique satisfiability is polynomial. Furthermore, if $\mathbf{P} \neq \mathbf{NP}$, there can be no polynomial parsimonious reduction from satisfiable Boolean formulas to this set, for this would contradict Theorem 3.5(2).

Now we consider minimization for very simple Boolean formulas. We show that the problem is "essentially at least as hard as" **SAT**. Since minimization is not a language recognition problem, this characterization cannot be expressed in tems of **SAT**-hard (npolygon, n). However the principle is the same. Any solution to the minimization problem can be used to solve some **SAT**-hard (npolylogn, n) problem in essentially the same time.

THEOREM 3.7. *Consider the problem of finding the minimal Boolean formula equivalent to a monotone formula in which no variable occurs more than twice. Let* $T(n)$ *be any increasing function such that, for all* k, $T(n) \geq n(\log n)^k$ *for almost all* n. *Suppose that* $T(\|w\|)$ *bounds above the deterministic time complexity of this problem in terms of formula size. Then* **SAT** $\in$ Dtime $(T(c\|w\|))$ *for some constant* c.

Proof. We will show how a minimization procedure can be used to solve problem (1) of Theorem 3.3. Let F and G be monotone Boolean formulas in which no variable occurs more than once in F or more than once in G. Let z be a variable that is not in F or G and consider the formula $H = (F \wedge z) \vee G$. The value of H is independent of z if and only if $F \leqq G$. But the minimum formula for H will have variable z if and only if H depends on z. Therefore $F \leqq G$ can be solved by scanning the minimum formula for H for the presence of variable z. □

We note that the proof of Theorem 3.7 goes through if the statement of the theorem begins "Consider the problem of finding the minimal *monotone* Boolean formula . . . ."

The results in this section are close to the best possible in that further simplifications almost always give problems that are known to be polynomial.

Finally, direct analogues of the theorems in this section hold for Boolean formulas involving operators other than *and*, *or*, and *not*. The next result lists a number of cases where hard formulas can be constructed using variables which appear no more than twice.

COROLLARY 3.8. *The satisfiability and tautology problems are* **SAT**-complete (npolylogn, n) *for Boolean formulas F that involve only variables, parentheses, and one of the following five possibilities*:

(i) *The nand operator* $|$ *and the constant* **1**,
(ii) *The nor operator* $\downarrow$ *and the constant* **0**,
(iii) *The implication operators* $\Rightarrow$ *and the operator not*,
(iv) *The implication operator* $\Rightarrow$ *and the constant* **0**, *or*
(v) *The exclusive or operator* $\oplus$, *the and operator* $\bullet$, *and the constant* **1**.

This statement remains true when F is restricted so that no variable occurs more than two times in F. Furthermore, the $\leqq$ *problem is* **SAT**-complete (npolylogn, n) *for pairs F, G of such Boolean formulas such that no variable occurs more than once in F and more than once in G.*

Proof. Recall the following logical identities:

(1) *not* $a = a|\mathbf{1} = a\downarrow\mathbf{0} = a \Rightarrow \mathbf{0} = a \oplus \mathbf{1}$,
(2) *a or* $b = (a|\mathbf{1})|(b|\mathbf{1}) = (a\downarrow b)\downarrow\mathbf{0} =$ *not* $a \Rightarrow b = (a \Rightarrow \mathbf{0}) \Rightarrow b$,
(3) *a and* $b = (a|b)|\mathbf{1} = (a\downarrow\mathbf{0})\downarrow(b\downarrow\mathbf{0}) =$ *not* $(a \Rightarrow$ *not* $b)$, and
(4) $a \Rightarrow b = \mathbf{1} \oplus [(\mathbf{1} \oplus b) \bullet a]$.

Because the quantities a and b appear once on each side of these identities, the identies can be used to linearly transform expressions written with {*and*, *or*, *not*} into expressions of the five types described in the corollary. Furthermore, this transformation will preserve the number of occurrences of each variable. The corollary then follows directly from Theorems 3.2 and 3.3 □

4. Applications to lattices, logic, and circuits. We use the results and proof techniques of § 3 to show that a number of basic problems are **SAT**-hard (npolylogn, n) and/or **SAT**-complete (npolylogn, n) for a wide collection of lattices. These problems include the $\leqq$, equivalence, and minimization problems for formulas, and the satisfiability and unique satisfiability problems for systems of equations. These lattices include all finite, finite-depth, atomic, and distributive lattices. Such lattices appear throughout

discrete mathematics and computer science, especially in logic [36], [43], [44], combinatorics and geometry [2], [7], [53], and the design, analysis, and testing of combinational logic circuits [11], [12], [19]-[21], [38], [46], [50]. Several applications are presented to logic and to circuit analysis and testing.

4.1. SAT-hard and -complete problems for lattices. We first show that very close analogues of the complexity results in § 3 for monotone Boolean formulas hold for each finite lattice.

THEOREM 4.1. *Let* $\mathbf{L}=(S, \vee, \wedge)$ *be a finite lattice. Let* R *be a representation of* $\mathbf{L}$. *Consider the problems of Fig.* 1 *for* $\mathbf{L}$ *and* R.

(1) *Problems* 1-10 *of Fig.* 1 *are* **SAT**-complete (npolylogn, n).

(2) *Problems* 11 *and* 12 *of Fig.* 1 *are* Turing-**SAT**-complete (npolylogn, n).

(3) *Let* $T(n)$ *be any increasing function such that, for all* k, $T(n) \geqq n(\log n)^k$ *for almost all* n. *Suppose that* $T(\|w\|)$ *bounds above the deterministic time complexity of Problem* 13 *of Fig.* 1. *Then* $\mathbf{SAT} \in \mathrm{Dtime}\,(T(c\|w\|))$ *for some constant* c.

Proof. Let $\mathbf{L}$ and R be as specified in the statement of the theorem. The proof has two parts.

Part 1. Proof of indicated lower bounds. It suffices to prove that Problems 2, 4, 6, 8, and 11 of Fig. 1 are **SAT**-hard (npolylogn, n) and that claim (3) of the statement of the theorem holds for Problem 13 of Fig. 1. Let $a \in S$ be an atom of $\mathbf{L}$. Let $\mathbf{a}$ be the constant symbol of $\mathscr{R}$ denoting the element a. Let F and G be monotone Boolean formulas such that

No variable occurs more than once in F and more than once in G.

Let F' and G' be the formulas on $\mathbf{L}$ and $\mathscr{R}$ that result from F and from G, respectively, by replacing

Each occurrence of *and* by $\wedge$,
Each occurrence of *or* by $\vee$, and
Each occurrence of a variable, say x, by $(x \wedge \mathbf{a})$.

---

1. The $\leqq$-problem for formulas F and G on $\mathbf{L}$ and $\mathscr{R}$.
2. Problem 1 restricted to the case where no variable appears more than once in F or more than once in G.
3. The equivalence problem for formulas on $\mathbf{L}$ and $\mathscr{R}$.
4. Problem 3 restricted to the case where no variable appears more than once in F or twice in G.
5. Determining if a system of equations on $\mathbf{L}$ and $\mathscr{R}$ has a solution.
6. Problem 5 restricted to the case of two equations in which no variable appears more than twice, once in each equation.
7. Determining if a set of equations $f_1 = g_1, \cdots, f_k = g_k$ implies an equation $f = g$ for formulas on $\mathbf{L}$ and $\mathscr{R}$.
8. Problem 7 restricted to the case $f_1 = c_1$ implies $f = c$ on $\mathbf{L}$ where c_1 and c are constants of $\mathscr{R}$ and no variable occurs more than once in f_1 or once in f.
9. Determining if a Boolean combination of equations of the form $f = g$ where f and g are formulas on $\mathbf{L}$ and $\mathscr{R}$, is satisfiable on $\mathbf{L}$ and $\mathscr{R}$.
10. Determinig if a Boolean combination of equation of the form $f = g$, where f and g are formulas on $\mathbf{L}$ and $\mathscr{R}$, is true for $\mathbf{L}$.
11. Determining if a system of equations on $\mathbf{L}$ and $\mathscr{R}$ has a unique solution.
12. Problem 11, even if the system has only three equations and no variable occurs more than four times in the system.
13. Given a formula F on $\mathbf{L}$ and in which no variable occurs more than twice, finding an equivalence formula H on $\mathbf{L}$ and $\mathscr{R}$ of minimal size.

FIG 1. *Problems that are hard for finite lattices.*

Then, the following are equivalent.

(a) $F \leqq G$.

(b) $F' \leqq G'$ on **L**.

(c) $F' \wedge G' = G'$ on **L**.

(d) The system of two equations on **L** and $\mathscr{R}$

$$F' = \mathbf{a} \quad \text{and} \quad G' = 0$$

has no solution.

(e) $G' = 0$ implies $F' = 0$ on **L**.

(f) Let $x_1, \cdots$, and x_n be the variables occuring in F or in G. Let $y_1, \cdots$, and y_n be n additional variables. The system of three equations on **L** and $\mathscr{R}$

$$((x_1 \wedge y_1) \vee \cdots \vee (x_n \wedge y_n)) \vee G = 0,$$

$$F \vee (y_1 \wedge \cdots \wedge y_n) = \mathbf{a},$$

$$(x_1 \vee y_1) \wedge \cdots \wedge (x_n \vee y_n) = \mathbf{a}$$

has a unique solution.

(g) Let z be a variable not occurring in F' or in G'. Let H' be the formula $(F' \wedge z \wedge \mathbf{a}) \vee G'$. A formula on **L** and $\mathscr{R}$ equivalent to H' of minimal size does not have an occurrence of the variable symbol z in it.

This equivalence is obtained by arguments closely similar to those of the proofs of the Theorems 3.3, 3.5, 3.7, and Corollary 3.4. To see this, it suffices to observe that

$$\{b \wedge a \mid b \in S\} = \{0, a\}$$

and

> Letting $\vee'$ and $\wedge'$ be the restrictions of $\vee$ and $\wedge$ of **L**, respectively, to $\{0, a\}$, the structures $(\{0, a\}, \vee', \wedge')$ and **BIN** are isomorphic distributive lattices.

In part (f), the third equation restricts the x_i and y_i to $\{0, a\}$. Thus by Theorem 3.3, Problems 2, 4, 6, 8, and 11 of Fig. 1 are each **SAT**-hard (npolylogn, n) and claim (3) of the statement of Theorem 4.1 holds.

Part 2. Proof of indicated upper bound. To prove the upper bounds on Problems 1-10 of Fig. 1 claimed by the theorem it suffices to prove that Problem 9 of Fig. 1 is npolylogn time and linear size reducible to **SAT**. The reduction is a fairly direct extension of that of the proof of Proposiion 2.4 and is illustrated in Fig. 2. The reduction of equations on L to **SAT** uses well-known encodings of finite structures into the two-element Boolean algebra. □

The $\leqq$, equivalence, and minimization problems for formulas on a finite lattice were shown to be **coNP**-hard in [26]. The reductions used to prove this are highly nonlinear in size. For example, for distributive lattices the reductions are already of size $\Theta(\|w\|^2)$. For nondistributive lattices, the reductions are significantly less size efficient.

Part (1) of the proof of Theorem 4.1 can be easily generalized so as to apply to many additional lattices as follows. Let $\mathbf{L} = (S, \vee, \wedge)$ be a lattice with elements $b, a \in S$ such that a covers b. Then, $\{(c \vee b) \wedge a \mid c \in S\} = \{b, a\}$. Also letting $\vee'$ and $\wedge'$ be the restrictions of $\vee$ of $\wedge$ of **L**, respectively, to $\{b, a\}$, the structures $(\{b, a\}, \vee', \wedge')$ and **BIN** are isomorphic distributive lattices. Let **b** and **a** be distinct constant symbols denoting b and a, respectively. Then, Problems 1-12 of Fig.1 are **SAT**-hard (npolylogn, n) for formulas and for systems of equations on **L**, where the only allowable constant symbols are **b** and **a**. The minimization problem for such formulas on **L** is also "as hard as" **SAT** in the sense of claim (3) of Theorem 4.1.

Boolean combination of equations:

$$(\sim(f_1 = g_1) \text{ or } (f_2 = g_2)) \text{ and } ((f_1 = g_3) \text{ or } \sim(f_2 = g_3))$$

3CNF formula for equation $f = g$:

$$(\bar{v}_f \text{ or } v_g) \text{ and } (v_f \text{ or } \bar{v}_g) \text{ and } \{\text{3CNF formula for } f\} \text{ and } \{\text{3CNF formula for } g\}$$

3CNF formula for the Boolean combination of equations:

$$\begin{aligned}
&\{\text{3CNF formula for } f_1 = g_1\} \text{ and}\\
&\{\text{3CNF formula for } f_2 = g_2\} \text{ and}\\
&\{\text{3CNF formula for } f_1 = g_3\} \text{ and}\\
&\{\text{3CNF formula for } f_2 = g_3\} \text{ and}\\
&\{\text{3CNF formula for } w_1 \equiv \sim(v_{f_1} \equiv v_{g_1})\} \text{ and}\\
&\{\text{3CNF formula for } w_2 \equiv (v_{f_2} \equiv v_{g_2})\} \text{ and}\\
&\{\text{3CNF formula for } w_3 \equiv (v_{f_1} \equiv v_{g_3})\} \text{ and}\\
&\{\text{3CNF formula for } w_4 \equiv \sim(v_{f_2} \equiv v_{g_3})\} \text{ and}\\
&\{\text{3CNF formula for } (w_1 \text{ or } w_2) \text{ and } (w_3 \text{ or } w_4)\}
\end{aligned}$$

FIG. 2. *Sample reduction: Boolean combinations of equations to* 3CNF *formulas.*

Finally, the generalized satisfiability and tautology problems, for a formula F on a lattice **L** with 0 and **1**, are the problems of determining

If there is an assignment **v** of values from the domain of **L** to the variables of F such that $\mathbf{V}[F] = 1$,

and if, for all assignments v of values from the domain of **L** to the variables of F,

$$\mathbf{v}[F] = 1.$$

Both problems are decidable deterministically in polynomial time, whenever constant expressions on **L** can be evaluated deterministically in polynomial time (e.g., **L** is a finite lattice). This is easily seen by noting the folllowing. Let $\mathbf{v}_1$ and $\mathbf{v}_2$ be the assignments of values from **L** to the variables of a formula F on **L** such that, for all variables x, $\mathbf{v}_1[x] = 0$ and $\mathbf{v}_2[x] = 1$. Then, $F = 1$ on **L** if and only if $\mathbf{v}_1[F] = 1$ on **L**. Also, there is an assignment **v** of values from **L** to the variables of F such that $\mathbf{v}[F] = 1$ if and only if $\mathbf{v}_2[F] = 1$.

4.2. Distributive lattices with an application to logic. By Theorem 1.4 the lower bounds of Theorem 4.1 also hold for each distributive lattice. In particular, Problems 1-4 of Fig. 1 are **SAT**-complete (npolylogn, n) for constant-free formulas on any distributive lattice. In the next two propositions, we show how each distributive lattice with 1 can naturally be extended so as to have a **SAT**-hard (npolylogn, n) generalized tautology or generalized satisfiability problem. In the first proposition, the extension is obtained by appending an "implication" operator such that $A \Rightarrow B$ means "B is more true than A." In the second, we append a "negative" operator such that some lattice element represents "not true."

PROPOSITION 4.2. *Let* $\mathbf{L}' = (S, \vee, \wedge, \Rightarrow)$ *be a nondegenerate algebraic structure such that*

(i) *The structure* $\mathbf{L} = (S, \vee, \wedge)$ *is a lattice;*

(ii) *There exists* 1 *in* S *such that, for all* $x \in S$, $x \leqq 1$ *on* **L**; *and*

(iii) *The operator* $\Rightarrow$ *is binary and, for all* $x, y \in S$, $(x \Rightarrow y) = 1$ *on* $\mathbf{L}'$ *if and only if* $x \leqq y$ *on* **L**.

Then, the set

$$\{(F, G) \mid F \text{ and } G \text{ are formulas on } \mathbf{L} \text{ such that } F \leqq G \text{ on } \mathbf{L}\}$$

is linear size reducible to the set

$$\{(F \Rightarrow G) \mid F \text{ and } G \text{ are formulas on } \mathbf{L};\ \text{and } (F \Rightarrow G) = 1 \text{ on } \mathbf{L}'\}.$$

In particular, if **L** *is a distributive lattice, then the set*

$$\{(F \Rightarrow G) \mid F \text{ and } G \text{ are constant-free formulas on } \mathbf{L} \text{ such that no variable occurs more than once in } F \text{ and more than once in } G;\ \text{and } (F \Rightarrow G) = 1 \text{ on } \mathbf{L}'\}$$

is **SAT**-complete (npolylogn, n).

Proof. For arbitrary **L**, the conclusion follows immediately from (iii). For distributive **L**, the additional conclusion follows from (iii), Theorem 1.4, and Theorem 3.3. □

PROPOSITION 4.3. *Let* $\mathbf{L}' = (S, \vee, \wedge, \sim)$ *be a nondegenerate algebraic structure such that*

(i) *The structure* $\mathbf{L} = (S, \vee, \wedge)$ *is a distributive lattice,*

(ii) *There exists* 1 *in* S *such that , for all* $x \in S$, $x \leqq 1$ *on* **L**, *and*

(iii) *The operator* $\sim$ *is unary,* $\sim 1 \neq 1$ *on* $\mathbf{L}'$, *and there exists* $b \in S$ *for which* $\sim b = 1$ *on* $\mathbf{L}'$.

Then, the set

$$\{(F \wedge (\sim G)) \mid F \text{ and } G \text{ are constant-free formulas on } \mathbf{L} \text{ such that no variable occurs more than once in } F \text{ and more than once in } G;\ \text{and there exists an assignment } \mathbf{v} \text{ of values from } S \text{ to the variables such that } \mathbf{v}[(F \wedge (\sim G))] = 1 \text{ on } \mathbf{L}'\}$$

is **SAT**-complete (npolylogn, n).

Proof. By Theorems 1.4 and 3.3, it suffices to prove that

There exists an assignment $\mathbf{v}$ of values from S to the variables such that $\mathbf{v}[(F \wedge (\sim G))] = 1$ on $\mathbf{L}'$ if and only if it is not the case that $F \leqq G$ on **L**.

The proof consists of two cases.

Case 1. If $F \leqq G$ on **L**, then $\mathbf{v}[(F \wedge (\sim G))] = 1$ on $\mathbf{L}'$ implies that $\mathbf{v}[F] = \mathbf{v}[G] = \sim \mathbf{v}[G] = 1$ on $\mathbf{L}'$, contradicting (iii).

Case 2. Suppose it is not the case that $F \leqq G$ on **L**. By Theorem 1.4 it is not the case that $F \leqq G$ on **BIN**. Hence, it is not the case that $F \leqq G$ on the distributive lattice $(\{b, 1\}, \vee', \wedge')$ where $\vee'$ and $\wedge'$ are the restrictions of the operators $\vee$ and $\wedge$, respectively, to $\{b, 1\}$. Thus, there is an assignment $\mathbf{v}$ of values from $\{b, 1\}$, and hence from **S**, to the variables such that $\mathbf{v}[G] = b$ and $\mathbf{v}[F] = 1$. Hence, $\mathbf{v}[(F \wedge (\sim G))] = 1$ using (iii). □

A number of the lattice-theoretical models of propositional calculi studied in the literature of algebraic logic [7], [43], [44] are known to satisfy the conditions of Propositions 4.2 and/or 4.3 [43]. Thus, there are many formula theories such that Proposition 4.2 implies that the logical validity and/or decision problems are **SAT**-complete (npolylogn, n) for simple formulas. These theories include the propositional calculi of classical two-valued logic in the logical theories L, L_1, L_2, L_3, and L_4 in [36], of positive logic [23], of intuitionistic logic [22], the modal logic S_4 [32], and for $m \geqq 2$, the m-valued logic of Post [42]. Intuitively, these theories are in a class of theories where suitable *and*, *or*, and *implication* operators can be defined by suitable formulas and the axioms and theorems evaluate to "true" in all associated models. Formalizing this class of theories is beyond the scope of this paper.

4.3. Some applications to circuit analysis and testing. Theorem 3.3 has a number of immediate applications to circuit analysis and testing, including computing signal

probability [41], [40], computing signal reliability [39], determining the testability of stuck-at faults [19], [38], [46], [50], and detecting the presence of static hazards [11], [12], [16]. To apply the theorem, we first give some definitions and observe some equivalences.

Let F and G be Boolean formulas with principle connectives *and* and *or*, respectively, and let z be a variable not occurring in F or G. (The principle connectives do not really matter but they are drawn as *and* and *or* in Fig. 3(a).) Let the combinational circuits $C_1[F, G]$, $C_2[F, G]$, and $C_3[F, G]$ be constructed from fan-out free monotone circuits for F and for G as shown in Fig. 3.

Given a set of variables, we let eq be the probability distribution on assignments that result when each variable is independently assigned the value 1 with probability one half. For any predicate P, we let $pr_{eq}\{P\}$ be the probability that P is true if the

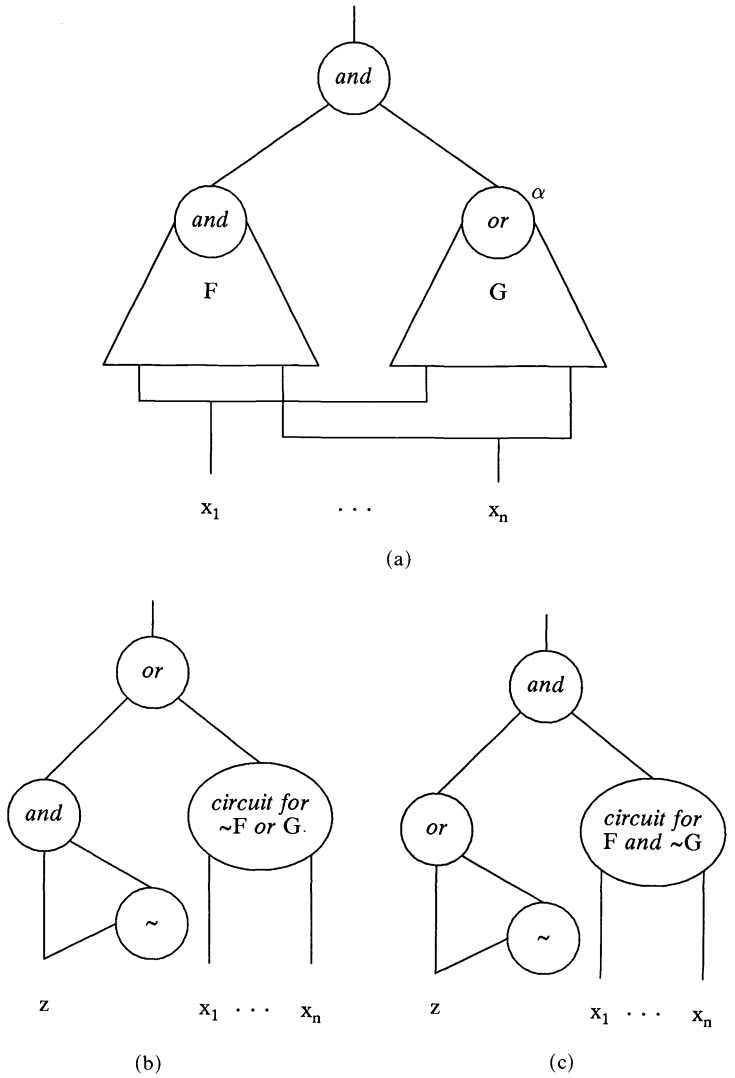


FIG. 3. *Circuits definitions for* § 4.3. (a) *The circuit* $C_1[F, G]$. (b) *The circuit* $C_2[F, G]$. (c) *The circuit* $C_3[F, G]$.

variables in P are assigned values randomly according to distribution eq.

Given the above definitions, the following statements are equivalent:

(i) $F \leqq G$.

(ii) $pr_{eq}\{F \text{ and } G = 1\} = pr_{eq}\{F = 1\}$.

(iii) $pr_{eq}\{F \text{ or } G = 1\} = pr_{eq}\{G = 1\}$.

(iv) $pr_{eq}\{G = 1 \mid F = 1\} = 1$.

(v) $pr_{eq}\{$the output of $C_1[F, G]$ is correct, when the gate labeled α is stuck-at-one and all other gates are correct$\} = 1$.

(vi) The gate labeled α in $C_1[F, G]$ is not testable for a stuck-at-one fault.

(vii) The circuit $C_2[F, G]$ does not have a static 0-hazard, when input z is indeterminant.

(viii) The circuit $C_3[F, G]$ does not have a static 1-hazard, when input z is indeterminant.

The equivalence of the first four statements is obvious. The others require some explanation since we are not giving the formal definitions of stuck-at faults and static hazards. The fault detection problem is to determine, by setting circuit inputs and observing circuit outputs, whether a specified circuit gate is performing properly. In Fig. 3(a), we would like to test if the gate labeled α always gives output one (i.e., is stuck at 1) instead of behaving (as it should) like an *or*-gate. To test this, we must set the variables so that the gate output should be 0 (i.e., G is false) and the output of circuit G is true. This cannot be done if and only if $F \leqq G$. With this explanation, the equivalence of (v) and (vi) to (i) should be apparent.

Static hazards are defined formally in terms of a three-valued logic with values 0, $\frac{1}{2}$, 1 where 0 and 1 behave as FALSE and TRUE and $\frac{1}{2}$ behaves as "undetermined." In Fig. 3(b), making z underdetermined (assigning z value $\frac{1}{2}$) causes (by definition) the output of the *and*-gate (which is input to the *or*-gate) to be undetermined. This indeterminancy will pass through the *or*-gate (by definition) if and only if the other input to the *or*-gate is 0 or $\frac{1}{2}$. But this can happen for determined assignments to $x_1 \cdots x_n$ if and only if $\sim F$ or G is not a tautology and hence not $F \leqq G$. The equivalence of (i) and (vii) should now be apparent and equivalence to (viii) becomes apparent with a dual argument.

From the above equivalences, the following result is immediate.

THEOREM 4.4. *Let F and G be monotone formulas satisfying the conditions of Theorem* 3.3. *Let the three-level monotone circuit $C_1[F, G]$, and the simple combinatorial circuits $C_2[F, G]$ and $C_3[F, G]$ be constructured from F and G as in Fig.* 3. *The following problems are* **SAT**-complete (npolylogn, n):

(1) *Determining if $pr_{eq}\{F \text{ and } G = 1\} = pr_{eq}\{F = 1\}$,*

(2) *Determining if $pr_{eq}\{F \text{ or } G = 1\} = pr_{eq}\{G = 1\}$,*

(3) *Determining if $pr_{eq}\{G = 1 \mid F = 1\} = 1$,*

(4) *Determining if $pr_{eq}\{$the output of $C_1[F, G]$ is correct, given that the gate labeled α is stuck-at-one and all other gates are operating correctly$\} = 1$,*

(5) *Determining if the gate labeled α in $C_1[F, G]$ is testable for a stuck-at-one fault,*

(6) *Determining if the circuit $C_2[F, G]$ has a static* 0*-hazard, when input variable z is indeterminant, and*

(7) *Determining if the circuit $C_3[F, G]$ has a static* 1*-hazard, when input variable z is indeterminant.* □

Conclusions (1)-(5) of Theorem 4.4 show that, even when two easy cases are combined, computing signal probability, computing the probabilities of joint or of conditional events, computing signal reliability, and determining the testability of stuck-at faults are "as hard as" the satisfiability problem for 3CNF formulas. (Recall

that $pr_{eq}\{H = 1\}$ can be computed deterministically in polynomial time, when H is a Boolean formula without repeated variables. Also, recall that the testability of single stuck-at faults can be determined deterministically in polynomial time, for combinational circuits without fanout.) Conclusion (5) strengthens the result of [50] that the testability problem for single stuck-at faults is **NP**-complete for three-level monotone circuits.

Finally, we point out how testing techniques in the literature can be interpreted in our algebraic context. From [46] it can be inferred that

> Determining the testability of a multiple stuck-at fault in a combinational circuit is npolylogn time and linear size reducible to determining if a system of equations over the four element Boolean algebra has a solution,

and from [12] it can be inferred that

> Determining if a combinational circuit has static 0- or 1-hazards, when a particular input variable is indeterminant, is npolylogn time and linear size reducible to determining if a system of equations on the three elements DeMorgan lattice L_3 has a solution.

In both cases, the later problem is **SAT**-complete (npolylogn, n). Thus, both determining the testability of single stuck-at faults and determining the presence of static 0- and 1-hazards in the very simple combinational circuits of the statement of Theorem 4.4 are "as hard as" the respective problems for arbitrary combinational circuits.

5. Applications to finite fields, modular arithmetic, binary decision diagrams, and program schemes. We use the results and proof techniques of § 3 to show that several basic problems are also **SAT**-hard (npolylogn, n), for finite fields, rings $\mathbf{Z}_k$ $(k \geqq 2)$ of integers modulo k, binary decision diagrams (bdds) [5], and monadic single variable program schemes [35]. Our new results strengthen and simply **NP**- and **coNP**-hardness, results, for rings in [29], [9], and [27] and for bdds and monadic single variable program schemes in [25] and [17]. Assuming $\mathbf{P} \neq \mathbf{NP}$, a number of the results obtained are "best" possible.

5.1. Finite fields and modular arithmetic. In [9] the equivalence problem is shown to be **coNP**-hard, for formulas on each finite field and on each ring $\mathbf{Z}_k$ $(k \geqq 2)$. Here, we use Theorem 3.2 to show, for each of these rings, that the equivalence problem is both **coNP**-hard and **SAT**-hard (npolylogn, n) for formulas involving only the operations $+$, $-$, $\bullet$, and exponentiation by constants in which no variable occurs more than two times.[1] As a corollary of the proof, we also show, for each of these rings **R**, that

> Determining if a system of equations on **R** in which no variable occurs more than once in each equation

is both **NP**-hard and **SAT**-hard (npolylogn, n). This last result strengthens results in [27].

[1] By exponentiation by constants, we mean that we are allowed to denote a formula $F \bullet \cdots \bullet F$ ($n \geqq 2$ F's) by F''. For such formulas F'', the number of occurrences of a variable in F'' equals the number of occurrences of the variable in F (rather than, n times the number of occurrences in the variable in F).

THEOREM 5.1. *The following problems are both* **coNP**-hard *and* **SAT**-hard (npolylogn, n):

(i) *For all finite fields* **F**, *determining if a formula on* **F**, *involving only variables, parentheses,* $+$, $-$, $\bullet$, *exponentiation by constants, and one in which no variable occurs more than two times, is equivalent to* 0 *on* **F**.

(ii) *For all* $k \geqq 2$, *determining if a formula on the ring* $\mathbf{Z}_k$, *involving only variables, parentheses,* $+$, $-$, $\bullet$, *exponentiation by constants, and one in which no variable occurs more than two times, is equivalent to* 0 *on* $\mathbf{Z}_k$.

Proof. (i). Let S be the domain of **F**. Let $k = |S|$. Recall that, for all $a \in S$, $a^{k-1} = 1$, if $a \neq 0$, and $a^{k-1} = 0$, if $a = 0$ [34]. Let f_0, f_1, f_2, and f_3 be the functions on S defined by

$$f_0(a) = a^{k-1}, \quad f_1(a) = 1 - a, \quad f_2(a, b) = 1 - [(1-a) \bullet (1-b)], \quad f_3(a, b) = a \bullet b.$$

Let $\boldsymbol{f}_1, \boldsymbol{f}_2$, and $\boldsymbol{f}_3$ be the restrictions of f_1, f_2, and f_3, respectively, to $\{0, 1\}$.

The structure $(\{0, 1\}, \mathbf{f}_2, \mathbf{f}_3, \mathbf{f}_1, 0, 1)$ is isomorphic to the two-element Boolean algebra. Thus, the claim of the theorem for (i) follows by a direct simulation of the proof of Theorem 3.2 by replacing each occurrence of a variable, say z, by z^{k-1}, each occurrence of *or* by $\mathbf{f}_2$, each occurrence of *and* by $\mathbf{f}_3$, and each occurrence of *not* by $\mathbf{f}_1$. Since the formulas for expressing $\mathbf{f}_1(a)$, $\mathbf{f}_2(a, b)$, and $\mathbf{f}_3(a, b)$ on **F** have the same number of occurrences of a and of b as the formulas *not a*, *a or b*, and *a and b*, respectively, this replacement can be accomplished in deterministic linear time.

(ii) The proof follows that of Corollary 4.3 of [9, pp. 897, 898]. Let S be the domain of $\mathbf{Z}_k$. There are two cases.

Case 1. $k = p^m$ for some integer $m \geqq 1$ where p is a prime. The proof is the same as that for (i) above except that the function f_0 on S is defined by $f_0(a) = a^{p^{m-1}(p-1)}$. Note that by Euler's theorem, $f_0(a) = 1$, if $p \nmid a$, and $f_0(a) = 0$, if $p \mid a$.

Case 2. $k = p^m n$ where $p \neq 2$ is a prime, m is an integer $\geqq 1$, and $p \nmid n$. By the Chinese Remainder Theorem, $\mathbf{Z}_k$ is isomorphic to $\mathbf{Z}_{p^m} \times \mathbf{Z}_n$, where the isomorphism I is given by, for all $a \in S$, $I(a) = (a_1, a_2)$ where $a_1 \equiv a \bmod p^m$ and $a_2 \equiv a \bmod n$. Let $A = \{x \mid 0 \leqq x < p^m n, \text{ and } p \mid x\}$, and let $\beta = I^{-1}((1, 0))$. Let f_0 be the function on S defined by $f_0(a) = (na)^{p^{m-1}(p-1)}$. If $a \in A$, then p divides a, and hence, $p^m n$ divides $f_0(a)$. Thus, $f_0(a) = 0$. If $a \in S - A$, then $\gcd(na, p^m) = 1$. Thus by Euler's theorem $(na)^{p^{m-1}(p-1)} \equiv 1 \bmod p^m$. Also, $(na)^{p^{m-1}(p-1)} \equiv 0 \bmod n$. Thus, $I((na)^{p^{m-1}(p-1)}) = (1, 0)$, and hence, $f_0(a) = \beta$. Let f_1, f_2, and f_3 be the functions on S defined by

$$f_1(a) = \beta - a, \quad f_2(a, b) = f_1(a + b), \quad f_3(a, b) = \beta - f_2(\beta - a, \beta - b).$$

Since $p \neq 2$, $2\beta \in S - A$. Thus, f_1 maps $\{0, \beta\}$ to $\{0, \beta\}$ and f_2 and f_3 map $\{0, \beta\} \times \{0, \beta\}$. Let f_1', f_2', and f_3' be the restrictions of f_1, f_2, and f_3, respectively, to $\{0, \beta\}$. Then, the structure $(\{0, \beta\}, f_2', f_3', f_1', 0, \beta)$ is isomorphic to the two element Boolean algebra. As in the proof for (i) above, the formulas for f_1, f_2, and f_3 do not have repeated variables and a linear time transformation of problems can be accomplished. □

COROLLARY 5.2. *Let* **R** *be any finite field or ring* $\mathbf{Z}_k$ $(k \geqq 2)$. *Then, determining if a system of two equations on* **R** *of the form*

$$f_1 = c_1, \qquad f_2 = c_3$$

has a solution on **R** *is both* **NP**-hard *and* **SAT**-hard (npolylogn, n), *where* f_1 *and* f_2 *are formulas on* **R** *involving only varibles, parentheses, exponentiation by constants, and one in which no variable occurs more than once in* f_1 *and more than once in* f_2 *and* c_1 *and* c_2 *are constant symbols denoting elements of* **R**.

Proof. In each case, the theorem follows by a direct simulation of the proof of Theorem 3.3 using the replacement given in the proof of Theorem 5.1. □

5.2. Binary decision diagrams and program schemes. The *Executability problem* (EP) for a class C of program schemes is the problem of determining, given a scheme S in C and a label λ of S, if there exists an interpretation I of S such that the statement labeled by λ in S is executed during the computation of S under I. In [25] the EP has been shown to be **NP**-complete for the class *Sw* of monadic single variable program schemes without loops consisting only of predicate tests and halt statements. Theorem 3.2 can easily be combined with the proof of [25] to prove the significantly stronger result that the EP is both **NP**-complete and **SAT**-hard (npolylogn, n), for the classes of program schemes S in *Sw* such that no predicate test occurs more than two times in S.

One easy and immediate corollary is the following.

THEOREM 5.3. *The isomorphism, strong equivalence, weak equivalence, containment, totality, and divergence problems are already* **SAT**-hard (npolylogn, n), *for monadic single variable program schemes S such that no predicate test occurs more than two times in S.*

Any monadic single variable program scheme in which no predicate test occurs more than once is *free.* Thus Theorem 3.3 yields a simple, immediate, and direct proof of the following result.

THEOREM 5.4. *The weak equivalence and containment problems are* **coNP**-complete *and are* **SAT**-hard (npolylogn, n), *for the free monadic single variable program schemes.*

Proof. Let F and G be monotone Boolean formulas, each without repeated variables. As shown in Fig. 4, the monadic single variable program schemes S_F and S_G can be constructed from F and G, respectively, in deterministic npolylogn time. Clearly, the sizes of S_F and S_G are linearly bounded in the sizes of F and G, respectively. Since no variable is repeated in F or in G, no predicate test occurs more than once in S_F and more than once in S_f. Thus, S_F and S_G are both free. Let S'_F and S'_G be the free monadic single variable program schemes in Fig. 5. It can easily be seen that the following statements are equivalent:

(1) $F \leqq G$;

(2) For any interpretation I such that the statement labeled A in S'_F is executed during the computation of S'_F under I, the statement labeled A in S'_G is executed during the computation of S'_G under I;

(3) S'_F is weakly equivalent to S'_G; and

(4) S'_F is contained by S'_G. □

Each monadic single variable program scheme in *Sw* can also be viewed as a binary decision diagram. Thus a number of strengthened hardness results for bdds can be read off from Theorems 3.2 and 3.3 and the proofs of Theorems 5.3 and 5.4. For example, the following holds.

THEOREM 5.5. *The tautology, satisfiability, and equivalence problems are both* **coNP**-complete, *and* **SAT**-complete (npolylogn, n) *for* bdds *in which no variable occurs more than two times. Moreover, the* $\leqq$ *problem is both* **coNP**-complete *and* **SAT**-complete (npolylogn, n) *for* bdds *in which no variable occurs more than one time.* □

Finally, let $F(x_1, \cdots, x_n)$ be a Boolean formula denoted by a bdd D_F in which no variable occurs more than one time along any path. A straight-line program, to compute the value of $p\{F = 1\}$ from the values of $p\{x_i = 1\}$ $1 \leqq i \leqq n$ for any independent probability distribution p, can be constructed from D_F deterministically in polynomial time. Let F and G be two such formulas. Let p_F and p_G be the associated straight-line programs computed from bdds D_F and D_G. Then, p_F and p_G are equivalent for all

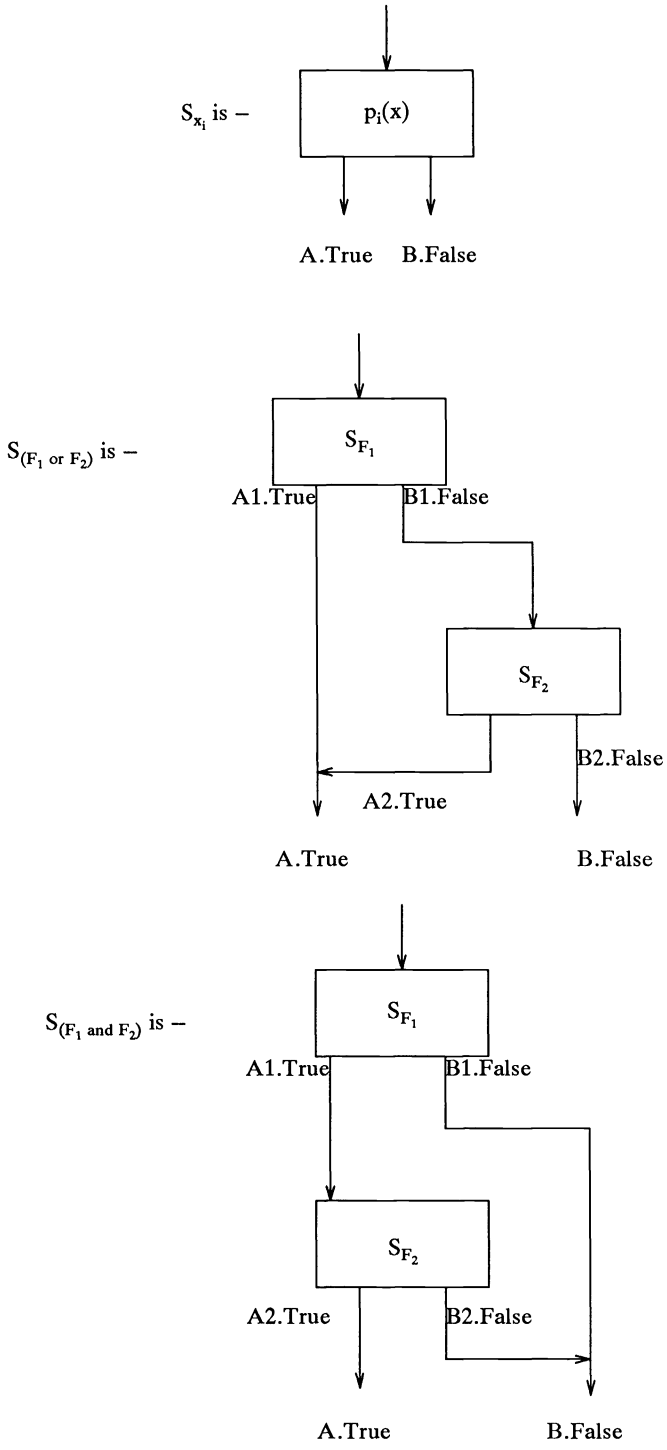


FIG. 4. *Program schemes for proof of Theorem* 5.3.

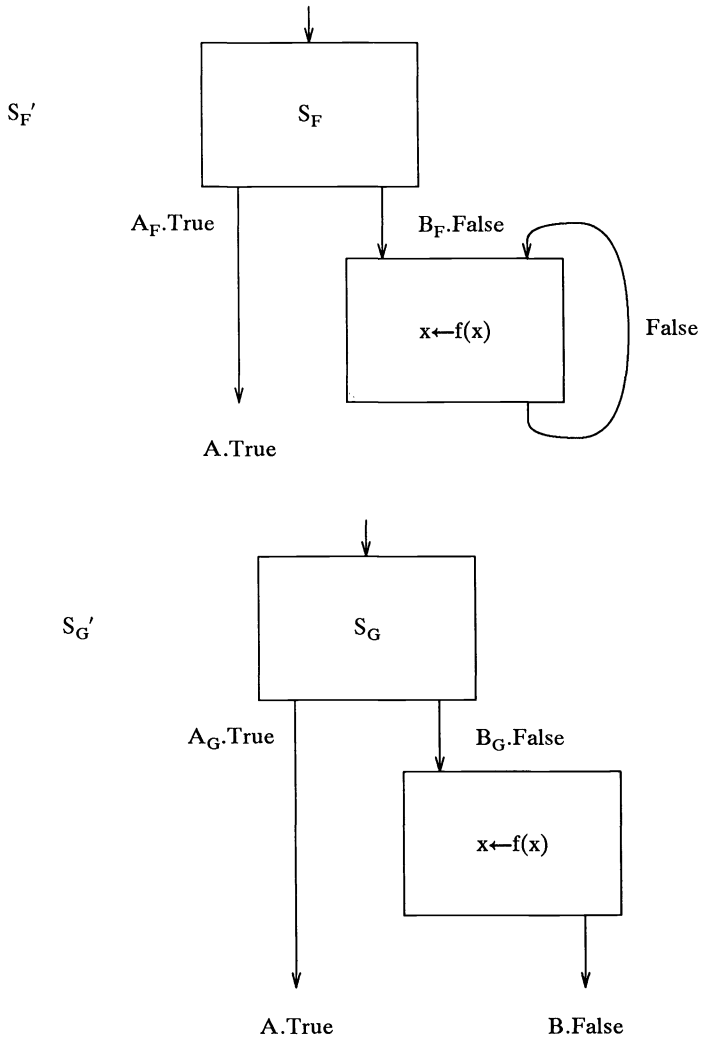


FIG. 5. *Program schemes for proof of Theorem* 5.4.

assignments of values from $\{x \text{ is a real} \mid 0 \leqq x \leqq 1\}$ to their variables if and only if $F = G$ if and only if p_F and p_G are equivalent for all assignments of values from the reals to their variables. Using the **RP** algorithm in [29] for the Inequivalence Problem for straight-line on infinite integer domains, we obtain an alternative proof for the following theorem from [10].

THEOREM 5.6. *There are* **RP** *algorithms for the inequivalence problem for* bdds *in which no variable occurs more than once along a path and for the strong equivalence problem* [35] *for free monadic single variable program schemes.*

6. Conclusion. The concepts of npolylogn time and linear size reducibility, **SAT**-hard (npolylogn, n), and **SAT**-completeness (npolylogn, n) have been introduced. Each **SAT**-hard (npolylogn, n) problem has been shown to require essentially as much deterministic time as **SAT**; and each **SAT**-complete (npolylogn, n) problem has been shown to require essentially the same deterministic time as **SAT**.

Extending our earlier work in [28], we have proved that the $\leqq$, satisfiability, tautology, unique satisfiability, equivalence, and minimization problems are already **SAT**-complete (npolylogn, n), for very simple Boolean formulas and systems of Boolean equations. In particular in Theorem 3.3, the $\leqq$ problem has been shown to be **SAT**-complete (npolylogn, n), for very simple monotone Boolean formulas F and G such that no variable occurs more than once in F or more than once in G. This problem, or equivalent variants of it, has been shown to be directly and naturally npolylogn time and linear size reducible to a number of problems for lattices, logics, combinatorial circuits, finite fields, modular arithmetic, monadic single variable program schemes, and binary decision diagrams. Thus, each of these additional problems is also **SAT**-hard (npolylogn, n).

Assuming $\mathbf{P} \neq \mathbf{NP}$, a number of the hardness results of this paper are "best" possible. In [13] it is shown that there is a deterministic polynomial time algorithm to convert a Boolean formula

> Involving only variables, parentheses, the operators *or*, *and*, *not*, and $\oplus$, and the constants 0 and 1 in which no variable occurs more than once

into an equivalent ordered bdd [17]. In [17] the equivalence problem for ordered bdds is shown to be decidable deterministically in polynomial time. Thus, the equivalence problem is also decidable deterministically in polynomial time, for pairs of Boolean formulas (F, G)

> Involving only variables, parentheses, the operators *or*, *and*, *not*, $\oplus$, $\Rightarrow$, $|$, $\downarrow$, and $\equiv$, and the constants 0 and 1 in which no variable occurs more than once in F and more than once in G.

(Contrast this with Corollaries 3.4 and 3.8.) Moreover, the satisfiability problem is decidable deterministically in polynomial time, for systems of equations of the form $F = c$, where F is such a Boolean formula, $c \in \{0, 1\}$, and no variable occurs more than once in the system. (Contrast this with Theorem 3.3(5).) For Boolean formulas involving only the operators *or*, *and*, and *not*, more can be said. Namely, two such formulas in negation normal form (i.e., *not*s are applied only to variables) are equivalent if and only if they are identical up to commutativity and associativity of *or* and of *and* [28]. One immediate corollary is that, for all lattices $\boldsymbol{L}$, the equivalence problem is decidable deterministically in polynomial time for constant-free formulas on $\mathbf{L}$ in which no variable occurs more than once. (Contrast this with Theorem 4.1(1).) Finally, we recall the remark in § 3 that the unique satisfiability result in Theorem 3.5 is best possible in that the same problem for two repetitions can be solved in polynomial time. As noted in § 3, this means (assuming $\mathbf{P} \neq \mathbf{NP}$) that there is no parsimonious reduction of the satisfiability problem for CNF formulas to the satisfiability problem for Boolean formulas in which no variable occurs more than twice.

Appendix. The purpose of this Appendix is to prove the following result mentioned in the discussion after Proposition 3.6.

THEOREM. *Let L be the set of pairs* (S, F) *such that*

(1) *S is a set of variables*;

(2) *F is a Boolean formula using operators* {*and, or, not*}, *constants* {TRUE, FALSE}, *variables from S, and parentheses*;

(3) *No variable appears more than twice in F*;

(4) *Only one assignment to variables in S make F true.*

There is a polynomial time algorithm for L.

Set S must be given as part of the problem so that we can represent the case where some variable of S occurs zero times in F. If some variable occurs zero times, then F is not uniquely satisfied.

There are certain simplifications that can be applied to any Boolean formula and which preserve properties (3) and (4) of the Theorem. DeMorgan's laws can be used so that formula F is monotone in literals. Formulas with constants can be simplified to formulas without constants (or to constant formulas). Pairs of the form $(S, x \wedge F)$ can be simplified to $(S-\{x\}, F_1)$ where F_1 is F with TRUE substituted for x. Finally, $(S, \bar{x} \wedge F)$ can be simplified to $(S-\{x\}, F_0)$ where F_0 is F with x replaced by FALSE.

The above simplifications can be applied repeatedly until the formula is a constant or a formula of the form

$$(*) \qquad (G_1 \vee H_1) \wedge \cdots \wedge (G_k \vee H_k)$$

for some $k \geqq 1$. The constant FALSE is never satisfiable and the constant TRUE is uniquely satisfiable if and only if the set of variables is empty. We thus only need a polynomial test for formulas of the form $(*)$.

If formula $(*)$ is uniquely satisfiable, there is an assignment that for each i, makes G_i or H_i true. We can assume without loss of generality that it is G_i, which is true. If some variable in S does not appear in any G_i, that variable can be changed without changing any of the G_i, and $(*)$ is not uniquely satisfiable. Thus each variable of S appears at least once in some G_i, and therefore the variables can appear at most once in the formula

$$(**) \qquad H_1 \wedge \cdots \wedge H_k,$$

and $(**)$ must have a satisfying assignment. Thus if $(*)$ is uniquely satisfiable, that assignment must make all the G_i and all the H_i true. Changing the value of a variable x in the assignment must make some $G_i \vee H_i$ false, and so x must appear in both G_i and H_i. From the above considerations, we conclude $(*)$ is uniquely satisfiable if and only if the following three conditions hold:

(i) All variables of S appear in $(*)$

(ii) For each i, G_i and H_i have the same variables

(iii) For each i, $G_i \vee H_i$ is uniquely satisfiable.

Conditions (i) and (ii) are easy to test in polynomial time and we get the main result if we can test condition (iii) in polynomial time. Each variable in formulas H_i and G_i occurs only once (by (3) and (ii)) and $G_i \vee H_i$ will have more than one solution if either G_i or H_i contains the operator *or*. We conclude that condition (iii) is equivalent to the following two conditions:

(iiia) Each G_i and H_i is the conjunction of literals;

(iiib) G_i and H_i have the same literals.

Both these conditions can be tested in polynomial time and the result is proved.

REFERENCES

[1] J. C. ABBOTT, *Sets, Lattices, and Boolean Algebras*, Allyn and Bacon, Boston, MA, 1969.

[2] L. ADLEMAN AND K. MANDERS, *Computational complexity of decision procedures for polynomials*, in Proc. 16th Annual IEEE Symposium on Foundations of Computer Science, Berkeley, CA, IEEE Computer Society, Washington, DC, 1975, pp. 169-177.

[3] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[4] A. V. AHO, R. SETHI, AND J. D. ULLMAN, *Compilers Principles, Techniques, and Tools*, Addison-Wesley, Reading, MA, 1986.

[5] S. B. AKERS, *Binary decision diagrams*, IEEE Trans. Comput., 27 (1978), pp. 509-516.

[6] M. BAUER, D. BRAND, M. FISCHER, A. MEYER, AND M. PATERSON, *A note on disjunctive tautologies*, SIGACT News, April 1973, pp. 17-20.
[7] G. BIRKHOFF, *Lattice Theory*, 3rd ed., American Mathematical Society, Providence, RI, 1967.
[8] A. BLASS AND YU. GUREVICH, *On the unique satisfiability problem*, Inform. and Control, 55 (1982), pp. 80-88.
[9] P. A. BLONIARZ, H. B. HUNT III, AND D. K. ROSENKRANTZ, *Algebraic structures with hard equivalence and minimization problems*, J. Assoc. Comput. Mach., 31 (1984), pp. 879-904.
[10] M. BLUM, A. K. CHANDRA, AND M. N. WEGMAN, *Equivalence of free boolean graphs can be decided probabilistically in polynomial time*, Inform. Process. Lett., 10 (1980), pp. 80-82.
[11] M. A. BREUER AND A. D. FRIEDMAN, *Diagnosis and Reliable Design of Digital Systems*, Computer Science Press, Rockville, MD, 1976.
[12] J. A. BRZOZOWSKI AND M. YOELI, *Digital Networks*, Prentice-Hall, Englewood Cliffs, NJ, 1976.
[13] S. CHAKRAVARTY, *On the testing, reliability analysis, and synthesis of combinatorial circuits*, Ph.D. dissertation, Departmentof Computer Science, State University of New York, Albany, NY, 1986.
[14] S. CHAKRAVARTY AND H. B. HUNT III, *On the generation of test vectors for multiple faults in digital circuits*, in Proc. 22nd Annual Allerton Conference on Communication, Control and Computing, October 1985, pp. 176-182.
[15] S. A. COOK, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151-158.
[16] E. B. EICHELBERGER, *Hazard detection in combinational and sequential switching circuits*, IBM J. Res. Develop., 9 (1965), pp. 90-99.
[17] S. FORTUNE, J. HOPCROFT, AND E. M. SCHMIDT, *The complexity of equivalence and containment for free single variable program schemes*, in Proc. 1978 Conference on Automata, Languages, and Programming, G. Ausiello and C. Bohm, eds., Lecture Notes in Computer Science, 62, Springer-Verlag, Berlin, New York, 1978, pp. 227-240.
[18] 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.
[19] P. GOEL, *An implicit enumeration algorithm to generate tests for combinational logic circuits*, IEEE Trans. Comput., 30 (1981), pp 215-222.
[20] J. P. HAYES, *A calculus for testing complex digital systems*, in Proc. Dig. 10th Fault-Tolerant Computing Symposium, Kyoto, Japan, 1980, pp. 115-120.
[21] ———, *Digital Simulation with multiple logic values*, IEEE Trans. Computer-Aided Design, (1986), pp. 274-283.
[22] A. HEYTING, *Intuitionism*, North-Holland, Amsterdam, 1959.
[23] D. HILBERT AND P. BERNAYS, *Grundlagen der Mathematik*, 1, Springer-Verlag, Berlin, New York, 1934.
[24] H. B. HUNT III, *Restricted set recognition problems and computational complexity*, Tech. Report 05-77, Center for Research in Computing Technology, Aiken Computation Laboratory, Harvard University, Cambridge, MA, 1977.
[25] H. B. HUNT III, R. L. CONSTABLE, AND S. SAHNI, *On the computational complexity of program scheme equivalence*, SIAM J. Comput., 9 (1980), pp. 349-416.
[26] H. B. HUNT, III, D. J. ROSENKRANTZ, AND P. A. BLONIARZ, *On the computational complexity of algebra on lattices*, SIAM J. Comput., 16 (1987), pp. 129-148.
[27] H. B. HUNT III AND R. E. STEARNS, *Nonlinear algebra and optimization on rings are "hard,"* SIAM J. Comput., 16 (1987), pp. 910-929.
[28] ———, *Monotone Boolean formulas, distributive lattices, and the complexities of logics, algebraic structures, and computation structures (preliminary report)*, in Proc. STACS86 3rd Annual Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 210, B. Monien and G. Vidal-Naquet, eds., Springer-Verlag, Berlin, New York, 1986, pp. 277-291.
[29] O. H. IBARRA AND S. MORAN, *Probabilistic algorithms for deciding equivalence of straight-line programs*, J. Assoc. Comput. Mach., 30 (1983), pp. 217-228.
[30] D. S. JOHNSON, *The NP-Completeness Column: An Ongoing Guide*, J. Algorithms, 6 (1985), pp. 291-305.
[31] S. C. KLEENE, *Introduction to Metamathematics*, D. Van Nostrand, Princeton, NJ, 1950.
[32] C. I. LEWIS AND C. H. LANGFORD, *Symbolic Logic*, Dover, New York, 1932.
[33] P. M. LEWIS, D. J. ROSENKRANTZ, AND R. E. STEARNS, *Compiler Design Theory*, Addison-Wesley, Reading, MA, 1976.
[34] S. MACLANE AND G. BIRKHOFF, *Algebra*, Macmillan, New York, 1967.
[35] Z. MANNA, *Mathematical Theory of Computation*, McGraw-Hill, New York, 1974.
[36] E. MENDELSON, *Introduction to Mathematical Logic*, 2nd ed., D. Van Nostrand, New York, 1979.

[37] M. MUKAIDONO, *A set of independent and complete axioms for a fuzzy algebra (Kleene algebra)*, in Proc. 11th IEEE International Symposium on Multiple-Valued Logic, 1981.

[38] P. MURTH, *A nine-valued circuit model for test generation*, IEEE Trans. Comput., 25 (1976), pp. 630–636.

[39] R. C. OGUS, *The probability of a correct output from a combinational circuit*, IEEE Trans. Comput., 24 (1975), pp. 534–544.

[40] K. P. PARKER AND E. J. MCCLUSKEY, *Analysis of faults using input signal probabilities*, IEEE Trans. Comput., 24 (1975), pp. 573–578.

[41] ———, *Probabilistic treatment of generalized combinational networks*, IEEE Trans. Comput., 24 (1975), pp. 668–670.

[42] E. L. POST, *Introduction to a general theory of elementary propositions*, Amer. J. Math., 43 (1921), pp. 165–185.

[43] H. RASIOWA, *An Algebraic Approach to Non-Classical Logics*, North-Holland, Amsterdam, 1974.

[44] H. RASIOWA AND R. SIKORSKI, *The Mathematics of Meta-Mathematics*, Panstwowe Wydawnictwo Naukowe, Warzawa, Poland, 1963.

[45] S. S. RAVI AND H. B. HUNT III, *An application of planar separator theorem to counting problems*, Inform. Process Lett. (1987), pp. 317–321.

[46] J. P. ROGTH, *Diagnosis of automata failures: A calculus and a method*, IBM J. Res. Develop., 10 (1966), pp. 278–291.

[47] J. B. SAXE, *Embeddability of weighted graphs in k-space is strongly* NP*-hard, in two papers on graph embedding problems*, Tech. Report CMU-CS-80-102, Department of Computer Science, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[48] R. E. STEARNS AND H. B. HUNT III, *On the complexity of the satisfiability problem and the structure of* NP, Tech. Report 86-21, Department of Computer Science, State University of New York, Albany, NY, 1986.

[49] ———, *On* SAT *and the relative complexities of* NP*-hard problems (extended abstract)*, Tech. Report 87-20, Department of Computer Science, State University of New York, Albany, NY, 1987.

[50] H. TOIDA AND S. FUJIWARA, *The complexity of fault detection problems for combinational logic circuits*, IEEE Trans. Comput., 31 (1982), pp. 555–560.

[51] C. A. TOVEY, *A simplified* NP*-complete satisfiability problem*, Discrete Appl. Math., 8 (1984), pp. 85–89.

[52] L. G. VALIANT AND V. V. VAZIRANI, NP *is as easy as detecting unique solutions*, in Proc. 17th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1985, pp. 458-463.

[53] D. J. A. WELSH, *Matroid Theory*, Academic Press, New York, 1976.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 71-77, February 1990

004

LOWER BOUNDS FOR THE STABLE MARRIAGE PROBLEM AND ITS VARIANTS*

CHENG NG† AND DANIEL S. HIRSCHBERG†

Abstract. In an instance of the stable marriage problem of size n, n men and n women, each participant ranks members of the opposite sex in order of preference. A *stable marriage* is a complete matching $M = \{(m_1, w_{i_1}), (m_2, w_{i_2}), \cdots, (m_n, w_{i_n})\}$ such that no unmatched man and woman prefer each other to their partners in M. There exists an efficient algorithm, due to Gale and Shapley, that finds a stable marriage for any given problem instance.

A pair (m_i, w_j) is *stable* if it is contained in some stable marriage. In this paper, the problem of determining whether an arbitrary pair is stable in a given problem instance is studied. It is shown that the problem has a lower bound of $\Omega(n^2)$ in the worst case. Hence, a previous known algorithm for the problem is asymptotically optimal.

As corollaries of these results, the lower bound of $\Omega(n^2)$ is established for several stable marriage related problems. Knuth, in his treatise on stable marriage, asks if there is an algorithm that finds a stable marriage in less than $\Theta(n^2)$ time. The results in this paper show that such an algorithm does not exist.

Key words. stable marriage problem, stable pair, analysis of algorithms, lower bounds

AMS(MOS) subject classifications. 68Q25, 90B99

Introduction. An instance of the stable marriage problem involves two disjoint sets of equal cardinality n, the men denoted by m_i's and women denoted by w_i's. Each individual ranks all members of the opposite sex in order of decreasing preference. A matching $M = \{(m_1, w_{i_1}), (m_2, w_{i_2}), \cdots, (m_n, w_{i_n})\}$ is a *stable marriage* if there does not exist an unmatched man-woman pair (m_i, w_j) such that both prefer each other to their partners in M. At least one stable marriage exists for any given problem instance. In most problem instances, there exists more than one stable marriage. Moreover, there are problem instances of size n where the number of stable marriages are exponential in n [IL86] [Kn76].

Gale and Shapley [GS62] first demonstrated that stable marriages exist for all problem instances and gave an algorithm that finds a stable marriage for any problem instance. The stable marriage obtained with the Gale-Shapley algorithm is *male-optimal*; that is, no man can receive a better match in any other stable marriage for the same problem instance. Moreover, by reversing the roles of men and women, the algorithm also finds the *female-optimal* stable marriage.

There are numerous expositions and analyses of the Gale-Shapley algorithm available in the literature [It78], [MW71], [Kn76]. The algorithm's worst-case asymptotic time complexity, $\Theta(n^2)$, is optimal for the stable marriage problem in the following sense. To input the description of a problem instance, which includes all preference rankings, requires $\Omega(n^2)$ time. However, the "computational" component (omitting time required for input) of the Gale-Shapley algorithm requires $O(n \log n)$ operations on the average [Wi72], despite its $\Theta(n^2)$ worst-case complexity.

It is interesting to investigate if there exists a faster algorithm that solves the problem under a model that ignores the input requirement. We shall elaborate on this model in the next section. In 1976, Knuth posed this question as one of twelve research problems in his treatise on stable marriage [Kn76]. Our main contribution in this paper

* Received by the editors June 27, 1988; accepted for publication (in revised form) March 28, 1989.
† Department of Information and Computer Science, University of California at Irvine, Irvine, California 92717.

is to show that such an algorithm does not exist; that the computational component of the stable marriage problem has a worst-case complexity of $\Omega(n^2)$. In a related problem, Gusfield [Gu87] asks if it is possible to determine in $o(n^2)$ time if an arbitrary complete matching is stable. We also answer this question in the negative by showing the lower bound of $\Omega(n^2)$ for this problem.

We have noted earlier that it is possible to have multiple stable marriages in a problem instance. We define a man-woman pair (m_i, w_j) *stable* if it is contained in some stable marriage. Consider the problem of determining whether an arbitrary pair is stable in a given problem instance. Gusfield [Gu87] provides an $O(n^2)$ algorithm that finds all stable pairs, and hence also solves the above problem. Our approach in this paper is first to show the $\Omega(n^2)$ lower bound for this problem. The other results follow as corollary.

1. Model of computation. In the introduction, we noted that our lower bound results must not depend on the time required to read the input for a problem instance. Hence, our model assumes that all participants' preferences are available in memory. It is useful to organize these preferences into two $n \times n$ integer matrices MP and WP such that the ith row of MP (WP) gives the preferences of m_i (w_i). For example, $MP[i, j] = k$ if m_i's jth preference is w_k.

For maximum generality, we also assume that two ranking matrices, denoted MR and WR, are available in memory. An entry in the men's ranking matrix, $MR[i, j]$, gives the ranking (position of preference) of w_j by m_i. Entries in WR, the women's ranking matrix, have similar interpretations.

The preference and ranking matrices are inverses of each other; for example, $MP[i, MR[i, j]] = j$ and $MR[i, MP[i, j]] = j$. Hence, the ranking matrices can be completely constructed from the preference matrices in $O(n^2)$ time. However, an algorithm may rely on the ranking matrices to determine quickly the ranking assigned to a participant by another of the opposite sex. Using the preference matrices to obtain this information can be slower because the algorithm has to search an entire row in the worst case.

We will use the notations MP, MR, WP, WR only when the problem instance associated with these matrices can be clearly determined from context. When there is a possibility of ambiguity, we use the notations MP_S, MR_S, WP_S, and WR_S, where S denotes a specific instance of the stable marriage problem.

Our lower bound is established by counting the number of times an algorithm must obtain information about the problem instance. In our model, such information is obtained with two types of queries. Given the identity of a participant and an integer i, the first type of query obtains the identity of his/her ith preference. Given two participants of opposite sex, the second type of query finds the ranking of the first participant in the second's preference. Each query can be accomplished in $O(1)$ time via a simple lookup of one of the four matrices.

2. The canonical instance. For every size n, our proofs are centered on a special instance of the stable marriage problem that we call the *canonical instance* and denote by C. An important characteristic of C is that the pair (m_n, w_n) is stable in it. However, there exists a large family of problem instances that differ only slightly yet sufficiently from C such that (m_n, w_n) is not stable in them. Later we will show how to construct such a problem instance which we call a *minimally noncanonical instance* and denote by $\sim C$.

We will show that before any algorithm can correctly determine that (m_n, w_n) is stable in C, it must make a certain minimum number of queries on the preference and

ranking matrices. Otherwise, it is possible to complete these matrices by giving appropriate values to the remaining entries that are not queried, and obtain a $\sim C$ that refutes the algorithm's correctness. This is due to the large number of possible $\sim C$'s, each derivable with only minor changes to C. Hence, the algorithm must make a large number of queries to eliminate all potential $\sim C$'s, supporting our lower bound claim.

We now define the women's preference matrix, WP_C. Entries in WP_C are defined by the function $WP_C[i, j] = j$, as illustrated in Fig. 1. Lemmas 1 and 2 give two properties of WP_C.

$$\begin{pmatrix} 1 & 2 & \dots & n \\ 1 & 2 & \dots & n \\ \vdots & \vdots & \ddots & \vdots \\ 1 & 2 & \dots & n \end{pmatrix}$$

FIG. 1. *Women's preference matrix, WP_C.*

LEMMA 1. *In a problem instance where WP_C is the women's preference matrix, the matching S constructed by the following rules is a stable marriage.*

(i) *When a woman receives a match, she is removed from the preference list of all remaining men.*

(ii) *Match m_1 with his highest preference.*

(iii) *After $m_1, m_2, \cdots, m_{i-1}$ are matched, m_i is matched with the highest preference remaining on his list.*

Proof. Rule (i) ensures that each woman is matched only once. Hence, S is a proper matching.

If m_i prefers w_j to his match in S, then by rule (iii), w_j is matched with m_k such that $k < i$. However, the preferences in WP_C show that w_j prefers m_k to m_i. Hence, m_i and w_j cannot destabilize S. □

LEMMA 2. *Regardless of the men's preferences, any problem instance that has WP_C as the women's preference matrix yields exactly one stable marriage.*

Proof. Any stable marriage is represented in WP_C by exactly one entry in each column. In particular, this is true of the male-optimal stable marriage S obtained by the Gale–Shapley algorithm.

Suppose there exists another stable marriage S'. For every matched pair (m_i, w_j) in S that has changed partners in S', m_i receives a less preferable partner in S' because S is male-optimal. Therefore, w_j receives a more preferable partner in S'. Otherwise, (m_i, w_j) is an unstable couple in S'.

Hence, every woman either has the same partner in both S and S', or she has a more preferable partner in S' than in S. According to WP_C, the subscript of each woman's partner decreases or stays the same. However, this requires that some column in WP_C be represented in S' by more than one entry. We conclude that S' does not exist. □

Lemma 1 gives us an algorithm that we will use in the proofs of Lemmas 3 and 4. The algorithm finds a stable marriage; it is shown by Lemma 2 to be the only one available.

We now describe the men's preference matrix, MP_C. Entries in MP_C fall into three groups. The first group, underlined in Fig. 2, includes the first row, last row, and tridiagonal entries of the remaining rows. The first and last rows consist of the integers 1 to n in increasing order. The tridiagonal entries of row i are the integers i, n, and

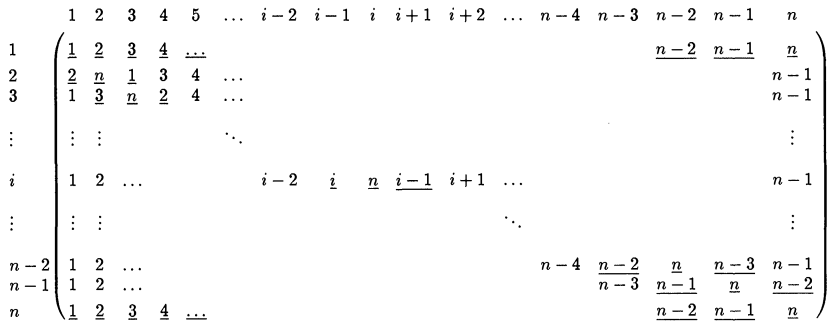


FIG. 2. *Men's preference matrix, MP_C.*

$i-1$ in that order. Of the remaining entries, the group left of the tridiagonals consists of integers 1 to $i-2$ in increasing order. The group right of the tridiagonals consi of integers $i+1$ to $n-1$ also arranged in increasing order.

LEMMA 3. *(m_n, w_n) is a stable pair in C.*

Proof. Apply the algorithm of Lemma 1 to C. Scanning the ith row of MP_C, note that all entries to the left of i have values less than i. These entries represent those women matched in previous rows. Hence, m_i's stable partner is w_i and $\{(m_1, w_1), (m_2, w_2), \cdots, (m_n, w_n)\}$ is a stable marriage in C. □

3. Obtaining noncanonical instances. Starting with C, we obtain a $\sim C$ by selecting a row i of MP_C such that $3 \leqq i \leqq n-2$, and exchanging two special entries, l and r, in that row. All entries left of the tridiagonal are candidates for l, but only those right of the tridiagonal with *values that differ from i by odd numbers* are candidates for r. Note that l is equal to its column number, and r's column number is $r+1$.

To formalize the above construction, we define, for each i, two sets of integers

$$L_i = \{x \mid 1 \leqq x \leqq i-2\}, \quad \text{and}$$

$$R_i = \{x \mid i+1 \leqq x \leqq n-1 \quad \text{and} \quad x \not\equiv i \pmod 2\}.$$

Then, for any i, l, and r satisfying $3 \leqq i \leqq n-2$, $l \in L_i$, and $r \in R_i$; we define $MP_{\sim C}[i, l] = r$ and $MP_{\sim C}[i, r+1] = l$. All other entries of $MP_{\sim C}$ and $WP_{\sim C}$ are equal to their corresponding entries in MP_C and WP_C.

LEMMA 4. *(m_n, w_n) is not a stable pair in $\sim C$.*

Proof. Apply the algorithm of Lemma 1 to $\sim C$. Figure 3 illustrates the stable marriage that results.

- m_k is matched with w_k for $1 \leqq k \leqq i-1$ because these rows are unchanged from C.
- m_i is matched with w_r. Note that $r \not\equiv i \pmod 2$, which guarantees that there is an even number of rows between row $i+1$ and row $r-1$ inclusive.
- For $i+1 \leqq k \leqq r-1$, m_k is matched with w_k if $k \not\equiv i \pmod 2$ and m_k is matched with w_{k-2} if $k \equiv i \pmod 2$. Note that m_{r-2} is matched with w_{r-2} and m_{r-1} is matched with w_{r-3}.

The above discussion shows that $w_1, w_2, \cdots, w_{i-1}$ are matched in rows 1 to $i-1$; $w_i, w_{i+1}, \cdots, w_{r-2}$ are matched in rows $i+1$ to $r-1$; and w_r is matched in row i. The subscripts of these women account for every entry left of the diagonal entry n in row r. Hence, m_r's partner is w_n.

$\sim C$ has only one stable marriage by Lemma 2. Since w_n is married to m_r and not m_n in this marriage, (m_n, w_n) is not a stable pair. □

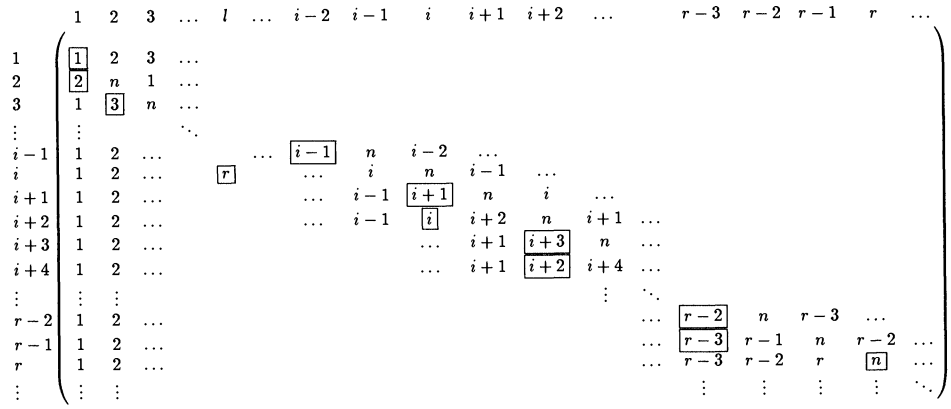


FIG. 3. *Stable marriage in* $\sim C$.

4. A counting argument. The construction of $\sim C$ is made possible by the exchange of appropriate l and r values. Until an algorithm has eliminated all possibilities of such exchanges, it cannot conclude correctly that it is dealing with the problem instance C. However, the large number of valid choices of i, l, and r gives us the following bound.

LEMMA 5. *If* $n=3k+4$ *for some integer* $k \geqq 1$, *the minimum number of queries needed to eliminate all possible constructions of* $\sim C$'s *is* $\frac{3}{2}k(k+1)$.

Proof. To eliminate row i from participating in the construction of a $\sim C$, the algorithm must query either all of L_i or all of R_i. To eliminate all possible constructions of $\sim C$'s, all rows must be eliminated.

$$|L_i| = i-2 \leqq k < \lceil (2k+1)/2 \rceil \leqq \lceil (n-i-1)/2 \rceil = |R_i|$$

$$\text{for } 3 \leqq i \leqq k+2, \qquad \text{and}$$

$$|L_i| = i-2 \geqq k+1 > \lceil 2k/2 \rceil \geqq \lceil (n-i-1)/2 \rceil = |R_i|$$

$$\text{for } k+3 \leqq i \leqq n-2.$$

Therefore the minimum number of queries needed

$$\begin{aligned}
&= \sum_{i=3}^{n-2} \min(|L_i|, |R_i|) \\
&= \sum_{i=3}^{k+2} |L_i| + \sum_{i=k+3}^{n-2} |R_i| \\
&= \sum_{i=3}^{k+2} (i-2) + \sum_{i=k+3}^{n-2} \lceil (n-i-1)/2 \rceil \\
&= \sum_{i=3}^{k+2} (i-2) + \sum_{i=k+3}^{3k+2} \lceil (3k+3-i)/2 \rceil \\
&= \sum_{j=1}^{k} j + \sum_{j=1}^{k} 2j \\
&= \tfrac{3}{2} k(k+1).
\end{aligned}$$

□

5. Lower bounds results. We are now ready to state our main result.

THEOREM. *Determining if an arbitrary pair is stable in a problem instance of size* n *requires* $\Omega(n^2)$ *time in the worst case.*

Proof. Without loss of generality, we may assume that $n = 3k+4$ for some integer $k \geqq 1$; otherwise, we extend the problem instance by adding the appropriate number of men and women.

By Lemmas 3 and 4, it is necessary to distinguish between C and $\sim C$ in order to determine if (m_n, w_n) is stable. By Lemma 5, any algorithm that distinguishes between C and $\sim C$ must make at least $\frac{3}{2}k(k+1) = \frac{3}{2}((n-4)/3)(((n-4)/3)+1)$ queries. Hence, the number of queries necessary is $\Omega(n^2)$. □

COROLLARY 1. *The asymptotic time complexity for determining if an arbitrary pair is stable in a problem instance of size n is* $\Theta(n^2)$.

Proof. The theorem provides an $\Omega(n^2)$ lower bound. Gusfield's algorithm provides an $O(n^2)$ upper bound. □

COROLLARY 2. *The asymptotic time complexity for finding a stable marriage in a given problem instance of size n is* $\Theta(n^2)$.

Proof. We noted earlier that the Gale–Shapley algorithm runs in $O(n^2)$ time. The only stable marriage in C is different from the only stable marriage in $\sim C$, and $\Omega(n^2)$ queries are required to distinguish between them. □

COROLLARY 3. *The asymptotic time complexity for determining if an arbitrary complete matching is a stable marriage in a given problem instance of size n is* $\Theta(n^2)$.

Proof. An obvious algorithm that solves this problem in $O(n^2)$ time exists. The matching $\{(m_1, w_1), (m_2, w_2), \cdots, (m_n, w_n)\}$ is a stable marriage in C but not in $\sim C$, and $\Omega(n^2)$ queries are required to distinguish between them. □

Historical note. The problem in Corollary 3 was raised by Gusfield [Gu87, p. 127]. He gives an algorithm that requires only $\frac{1}{2}n(n-1)+2n$ queries. By Lemma 5, we show that at least $\frac{1}{6}(n-4)(n-1)$ queries are needed.

6. Conclusions. We have shown that the lower bound of $\Omega(n^2)$ holds for three stable marriage-related problems. This lower bound is fundamental to stable marriage and holds for other variants of the stable marriage problem, including the following class of optimization problems. Given an instance of the stable marriage problem X, we define a real-valued function V, whose domain is the set of stable marriages in X. The problem of finding a stable marriage M that maximizes (or minimizes) $V(M)$ has a lower bound of $\Omega(n^2)$, by an argument similar to that of Corollary 2. By varying the definition of V, we can formulate different variants of the stable marriage problem. We give three such problems that have been studied in the literature.

Suppose (m_i, w_j) is a pair in a marriage. The *regret* of m_i is the ranking he gives to w_j, which equals $MR[i, j]$. Similarly, the regret of w_j equals $WR[j, i]$. The regret of a marriage M is the maximum regret among all the participants. The *minimum regret stable marriage problem* is to find a stable marriage with the minimum regret. Gusfield [Gu87] gives an algorithm that solves this problem in $O(n^2)$ time, which is asymptotically optimal.

The Gale–Shapley algorithm favors one set of participants heavily over the other. It is often desirable to obtain a stable marriage that treats both sexes more equitably. The *egalitarian stable marriage problem* is to find such a marriage M, one that minimizes $\sum_{(m_i, w_j) \in M} MR[i, j] + WR[j, i]$. An algorithm that solves the egalitarian problem in $O(n^3 \log n)$ time is given in [Fe89].

A weighted version of the egalitarian problem is the *optimal stable marriage problem.* In this problem, the rankings are replaced by general "unhappiness" functions $um(i, j)$ and $uw(j, i)$ for every possible pair (m_i, w_j). The goal is to find a marriage M that minimizes $\sum_{(m_i, w_j) \in M} um(i, j) + uw(j, i)$. An algorithm that solves the optimal problem in $O(n^4 \log n)$ time is given in [ILG87].

An important generalization of the stable marriage problem that has received substantial attention is the *stable roommate problem.* This problem involves only one set of participants. Using a similar definition for stability, the goal is to find an assignment (a partition of the participants into pairs) that is stable. For every variant of the stable marriage problem described in this paper, there is a corresponding stable roommate variant that is similarly defined. Moreover, the $\Omega(n^2)$ lower bound applies to these variants. This claim is supported by the observation that every instance of the stable marriage problem is also an instance of the stable roommate problem having the same solution structure. We refer readers to [Gu88, p. 767] for a general discussion of this relation.

Variants of the stable roommate problem that have $O(n^2)$ algorithms include the following: determining whether an arbitrary pair is stable [Gu89]; determining whether an assignment is stable; finding a stable assignment [Ir85]; and the minimum regret problem [Ir86]. Obviously, no asymptotic improvement is possible with these problems. Feder has shown that the egalitarian stable roommate problem—and by implication, the optimal stable roommate problem—is *NP*-complete [Fe89].

Acknowledgments. We thank Dan Gusfield and an anonymous referee for many helpful suggestions that have improved the presentation of this paper.

REFERENCES

[Fe89] T. FEDER, *A new fixed point approach for stable networks and stable marriages*, in Proc. 21st Symposium on Theory of Computing, Seattle, WA, 1989, pp. 513-522.

[GS62] D. GALE AND L. SHAPLEY, *College admissions and the stability of marriage*, Amer. Math. Monthly, 69 (1962), pp. 9-15.

[Gu87] D. GUSFIELD, *Three fast algorithms for four problems in stable marriage*, SIAM J. Comput., 16 (1987), pp. 111-128.

[Gu88] ———, *The structure of the stable roommate problem: Efficient representation and enumeration of all stable assignments*, SIAM J. Comput., 17 (1988), pp. 742-769.

[Gu89] ———, *Personal communications*, 1989.

[IL86] R. W. IRVING AND P. LEATHER, *The complexity of counting stable marriages*, SIAM J.Comput., 15 (1986), pp. 655-667.

[ILG87] R. W. IRVING, P. LEATHER, AND D. GUSFIELD, *An efficient algorithm for the "optimal" stable marriage*, J. Assoc. Comput. Mach., 34 (1987), pp. 532-543.

[Ir85] R. W. IRVING, *An efficient algorithm for the "stable roommates" problem*, J. Algorithms, 6 (1985), pp. 577-595.

[Ir86] ———, *On the stable room-mates problem*, Tech. Report CSC/86/R5, University of Glasgow, Glasgow, UK, 1986.

[It78] S. Y. ITOGA, *The upper bound for the stable marriage problem*, Oper. Res., 29 (1978), pp. 811-814.

[Kn76] D. E. KNUTH, *Mariages Stables*, Les Presses de L'Université de Montréal, Montréal, 1976.

[MW71] D. G. MCVITIE AND L. B. WILSON, *The stable marriage problem*, Comm. ACM, 14 (1971), pp. 486-492.

[Wi72] L. B. WILSON, *An analysis of the stable marriage assignment problem*, BIT, 12 (1972), pp. 569-575.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 78–97, February 1990

005

A REWRITING SYSTEM FOR CATEGORICAL COMBINATORS WITH MULTIPLE ARGUMENTS*

HIROFUMI YOKOUCHI† AND TERUO HIKITA‡

Abstract. Categorical combinators have been derived from the study of categorical semantics of lambda calculus, and it has been found that they may be used in implementation of functional languages. In this paper categorical combinators are extended so that functions with multiple arguments can be directly handled, thus making them more suitable for practical computation. A rewriting system named CCLMβ is formulated for these combinators. This system naturally corresponds to the type-free $\lambda\beta$-calculus. The relationship between these two systems is established, and as a result of this, the Church–Rosser property of CCLMβ is proved. A similar relationship is also established between the original CCLβ by Curien and the type-free $\lambda\beta$-calculus with product. Finally the embedding theorem of CCLMβ into CCLβ is shown.

Key words. categorical combinator, Church–Rosser property, combinator, functional programming, lambda calculus, rewriting system

AMS(MOS) subject classifications. 03B40, 68C01

1. Introduction.

Categorical models of lambda calculus have been extensively studied. See, e.g., [4], [5], [10]–[13], [18], [19] and the bibliographies therein. (See also [16].) Curien [4], [5] introduced categorical combinators from such categorical semantics of lambda calculus, and he formulated rewriting systems for them, such as CCLβ and CCL$\beta\eta$SP. In this paper, extending CCLβ, we propose a new kind of system called CCLMβ and establish a natural correspondence between CCLMβ and λ-calculus as rewriting systems.

It is well known that λ-calculus and Cartesian closed categories (CCC) are essentially the same. This correspondence involves explicit products in λ-calculus. Certainly the structure of products is expressed by λ-terms in the type-free λ-calculus, but, to give a strict correspondence between the two systems, the λ-calculus needs to have explicit products. For the computational aspect of λ-calculus, products are not so important; we may want to have a CCC-like structure that strictly corresponds to λ-calculus without products.

In the correspondence between λ-calculus and CCC, the product in CCC plays two different roles: one is handling variables in λ-terms, and the other is handling the product itself. Let M be a λ-term with free variables $z_1, \cdots, z_n$. Then M is regarded as an n-ary function. In CCC, M is interpreted as an arrow from the product of the objects corresponding to the free variables $z_1, \cdots, z_n$. Access of the free variables in M is represented by projections in CCC. The structure of the product in CCC is defined in a general form, and consequently the corresponding λ-calculus has the same general structure of product. This is why the λ-calculus needs explicit product for the strict correspondence between λ-calculus and CCC. If we separate these two uses of product, we may obtain a more natural correspondence between λ-calculus without product and a CCC-like structure.

Obtułowicz and Wiweger [17], based on functorial semantics of algebraic theory [14], introduced another kind of categorical models of λ-calculus, called Church

* Received by the editors November 3, 1987; accepted for publication (in revised form) April 22, 1989.
† IBM Research, Tokyo Research Laboratory, 5-19 Sanban-cho, Chiyoda-ku, Tokyo 102, Japan.
‡ Department of Mathematics, Tokyo Metropolitan University, 2-1-1 Fukazawa, Setagaya-ku, Tokyo 158, Japan. Present address, Department of Computer Science, Meiji University, 1-1-1 Higashi-Mita, Tama-ku, Kawasaki 214, Japan.

algebraic theory. They have shown that models of the pure type-free λ-calculus without product are essentially the same as Church algebraic theory. This result is a model theoretic one, that is, the correspondence between Church algebraic theory and λ-calculus is examined as equational systems. In this paper, however, we discuss syntactical rewriting systems for categorical structure of λ-calculus. Using Obtułowicz and Wiweger's idea, we extend categorical combinators by Curien and propose the rewriting system CCLMβ.

The key idea of CCLMβ is arities of functions. In CCLMβ, every term has its fixed arity, say n, and the term intuitively represents an n-ary function. In the original system CCLβ, the arity of the function represented by a term of CCLβ is determined according to the context in which the term appears. The system CCLMβ, moreover, has an operator that constructs n-tuples, and projection p_i^n that directly gives the ith member of an n-tuple. The projections are used in access of variables when a λ-term is interpreted in CCLβ.

We show a natural correspondence concerning reduction between the pure type-free $\lambda\beta$-calculus and CCLMβ. Through this correspondence, various properties of λ-calculus can be transferred to CCLMβ. In particular, we show that CCLMβ satisfies the Church–Rosser property using that of λ-calculus.

Incidentally, the systems of categorical combinators have a strong resemblance to the functional-style language FP of Backus [1]. Categorical combinators have been used in implementation of functional languages [3]. (See also [15].) Our system is suited for practical computation where multiple arguments prevail. Moreover, our system may be applied to partial computation (or often called partial evaluation), which is a method of computing a function with more than one argument by supplying values to only a specified part of the arguments. It has many applications such as compiler generation [7]. With the mechanism of arities, the operation of currying and application are naturally extended to "partial currying" and "partial application" in CCLMβ.

In § 2 the new categorical combinators are introduced, and the rewriting system CCLMβ is formulated for these combinators. In § 3 some derived combinators are introduced that will be useful in practical computation. In § 4 we briefly state the model theoretic aspects of the system. In § 5 the translation algorithms are introduced between CCLMβ and λ-calculus, and in § 6 theorems on the relationship concerning reduction between these two systems are established. In § 7, the Church–Rosser property of CCLMβ is proved using the results in § 6. In § 8, we deal with the original system CCLβ without arities and show similar results to those for CCLMβ. Finally, in § 9, we show that CCLMβ can be embedded into CCLβ.

We assume the reader has basic knowledge of λ-calculus (e.g., [2]). The acquaintance with categorical combinators [4], [5] is desirable, but this paper is self-contained and makes no use of previous results about them.

Recently, Curien [6] further extended the results of this paper.

2. Rewriting system CCLMβ. We extend the original categorical combinators by Curien [4] and introduce the formal system named CCLMβ. Before presenting its formal definition, we explain the intuitive meaning of the new combinators.

We design CCLMβ so that arities of terms are explicitly specified. First consider the operation $\circ$ that means function composition. Let f be an n-ary function, then the right-hand side of $f \circ (—)$ must be a multivalued function whose value is an n-tuple. We introduce an operation that constructs an n-tuple of functions. For n functions $f_1, \cdots, f_n$ of equal arity, say k, the angular bracket $\langle f_1, \cdots, f_n \rangle$ means a multivalued

function of arity k, the ith value of which is the value of f_i. Function composition is defined only in the form $f \circ \langle f_1, \cdots, f_n \rangle$ for an n-ary function f and an n-valued function $\langle f_1, \cdots, f_n \rangle$. We assume that every term is a single-valued function. By this assumption, $\langle f_1, \cdots, f_n \rangle$ itself is not a term, so that compound expressions such as $\langle g_1, \langle f_1, \cdots, f_n \rangle, g_2 \rangle$ are disallowed. Moreover, we introduce combinator p_i^n, $1 \leqq i \leqq n$, that means the ith projection of an n-tuple. The combinators p_i^n are extensions of Fst and Snd of the original categorical combinators.

For currying operation Λ, we specify the arities of functions. For $n \geqq 0$, the operation Λ_n applies to a function with $n+1$ arguments and means currying. More precisely, for a function f of arity $n+1$, $\Lambda_n(f)$ means the curried function whose arguments correspond to the first n arguments of f. Informally, $\Lambda_n(f)$ is represented as

$$\lambda \langle z_1, \cdots, z_n \rangle \cdot (\lambda x \cdot f(z_1, \cdots, z_n, x))$$

in a λ-calculus-like notation. To cope with this extension of the currying operation, App is also extended. In our definition, App receives two arguments, and the compositions of App appear only in the form $\mathrm{App} \circ \langle f_1, f_2 \rangle$. The first argument of App is regarded as a curried function and App applies the first argument to the second one. In a λ-calculus-like notation, App is represented as $\lambda \langle x, y \rangle \cdot xy$.

Now we formally give the definition of terms of CCLMβ.

DEFINITION. We define *terms* of CCLMβ with nonnegative integer called *arity*. A term with arity n is called n-ary term. For every constant its arity is uniquely specified. We assume that there are special constants: p_i^n of arity n for all pairs of n and i such that $n \geqq 1$ and $1 \leqq i \leqq n$, and App of arity 2. The other constants are called nonspecial constants. Then the terms of CCLMβ are defined as follows:

(1) Every constant is a term.
(2) For m, $n \geqq 0$, if F is an m-ary term and $G_1, \cdots, G_m$ are n-ary terms, then $F \circ \langle G_1, \cdots, G_m \rangle_n$ is an n-ary term.
(3) For $n \geqq 0$, if F is an $(n+1)$-ary term, then $\Lambda_n(F)$ is an n-ary term.

In the following, terms are denoted by F, G, F_1, etc. We write $F \equiv G$ when F and G are syntactically the same. We almost always omit the subscript n of $F \circ \langle G_1, \cdots, G_m \rangle_n$ and $\Lambda_n(F)$. Note that $F \circ \langle\,\rangle_n$ is a term of arity n for 0-ary term F. Note also that n-tuples in themselves are *not* terms; they always appear as part of composed terms.

Now we present the rewriting rules of the formal system CCLMβ.

DEFINITION. We define the binary relation $\rightarrow$ among the terms of CCLMβ by the following rules:

(1) $(F \circ \langle G_1, \cdots, G_m \rangle) \circ \langle H_1, \cdots, H_n \rangle$
$\rightarrow F \circ \langle G_1 \circ \langle H_1, \cdots, H_n \rangle, \cdots, G_m \circ \langle H_1, \cdots, H_n \rangle \rangle$.
(2) $p_i^n \circ \langle F_1, \cdots, F_n \rangle \rightarrow F_i$.
(3) $c \circ \langle p_1^n, \cdots, p_n^n \rangle \rightarrow c$, where c is an n-ary constant.
(4) $\Lambda(F) \circ \langle G_1, \cdots, G_n \rangle$
$\rightarrow \Lambda(F \circ \langle G_1 \circ \langle p_1^{k+1}, \cdots, p_k^{k+1} \rangle, \cdots, G_n \circ \langle p_1^{k+1}, \cdots, p_k^{k+1} \rangle, p_{k+1}^{k+1} \rangle)$,
where F is $(n+1)$-ary, and $G_1, \cdots, G_n$ are k-ary.
(5) $\mathrm{App} \circ \langle \Lambda(F), G \rangle \rightarrow F \circ \langle p_1^n, \cdots, p_n^n, G \rangle$, where F is $(n+1)$-ary, and G is n-ary.
(6) If $F \rightarrow F'$, then $F \circ \langle G_1, \cdots, G_m \rangle \rightarrow F' \circ \langle G_1, \cdots, G_m \rangle$.
(7) If $G_i \rightarrow G_i'$ for some i $(1 \leqq i \leqq m)$, then $F \circ \langle G_1, \cdots, G_i, \cdots, G_m \rangle \rightarrow$
$F \circ \langle G_1, \cdots, G_i', \cdots, G_m \rangle$.
(8) If $F \rightarrow F'$, then $\Lambda(F) \rightarrow \Lambda(F')$.

We denote by $\xrightarrow{*}$ the reflexive and transitive closure of $\rightarrow$. Note that arity is invariant under the relation $\rightarrow$ (and $\xrightarrow{*}$).

Example. We give an example of computation in CCLMβ. Let plus $(x, y, z) = x+y+z$ be a function with three arguments giving their sum. In CCLMβ, $\lambda xyz \cdot$ plus (x, y, z) is translated to the following (the translation algorithm will be given in § 5):

$$\Lambda(\Lambda(\Lambda(\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle))).$$

Now, we give only one value 2 to its first argument, and partially compute it using App. In the below, 2^n means the constant-valued function with n arguments giving 2 as its result:

$$\text{App} \circ \langle \Lambda(\Lambda(\Lambda(\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle))), 2^0\rangle$$

$$\rightarrow \Lambda(\Lambda(\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle)) \circ \langle 2^0\rangle \quad \text{(by rule (5))}$$

$$\rightarrow \Lambda(\Lambda(\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle)) \circ \langle 2^0 \circ \langle\,\rangle_1, p_1^1\rangle) \quad \text{(by rule (4))}$$

$$\rightarrow \Lambda(\Lambda(\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle) \circ \langle 2^1, p_1^1\rangle)$$

$$\rightarrow \Lambda(\Lambda((\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle) \circ \langle 2^1 \circ \langle p_1^2\rangle, p_1^1 \circ \langle p_1^2\rangle, p_2^2\rangle)) \quad \text{(by rules (4), (8))}$$

$$\xrightarrow{*} \Lambda(\Lambda((\text{plus} \circ \langle p_1^3, p_2^3, p_3^3\rangle) \circ \langle 2^2, p_1^2, p_2^2\rangle))$$

$$\xrightarrow{*} \Lambda(\Lambda(\text{plus} \circ \langle 2^2, p_1^2, p_2^2\rangle)).$$

3. Auxiliary combinators. For application of CCLMβ to practical situation, it is convenient to define more combinators. We introduce here several derived combinators and rewriting rules in a more general form. They are helpful to understand the mechanism of arities in CCLMβ, too. However, they will not be used in the succeeding sections.

DEFINITION.

(1) For $n \geqq 0$, define $\text{Id}^n \equiv \langle p_1^n, \cdots, p_n^n\rangle$.

(2) For $m \geqq 0$ and $n \geqq 0$, define $P^{m,n} \equiv \langle p_1^{m+n}, \cdots, p_m^{m+n}\rangle$.

(3) For an n-ary term F and m such that $1 \leqq m \leqq n$, define the $(n-m)$-ary term $\Lambda^m(F)$ by $\Lambda^1(F) \equiv \Lambda(F)$ and $\Lambda^{m+1}(F) \equiv \Lambda(\Lambda^m(F))$.

(4) For $m \geqq 1$ and n-ary terms $F, G_1, \cdots, G_m$, define the n-ary term $\text{APP}^m\{F, G_1, \cdots, G_m\}$ by

$$\text{APP}^1\{F, G_1\} \equiv \text{App} \circ \langle F, G_1\rangle,$$

$$\text{APP}^{m+1}\{F, G_1, \cdots, G_m, G_{m+1}\} \equiv \text{App} \circ \langle \text{APP}^m\{F, G_1, \cdots, G_m\}, G_{m+1}\rangle.$$

(5) For $m \geqq 1$, define $\text{App}^m \equiv \text{APP}^m\{p_1^{m+1}, p_2^{m+1}, \cdots, p_{m+1}^{m+1}\}$.

LEMMA 3.1.

(i) $\Lambda^m(\Lambda^n(F)) \equiv \Lambda^{m+n}(F)$.

(ii) $\text{APP}^m\{\text{APP}^l\{F, G_1, \cdots, G_l\}, H_1, \cdots, H_m\}$
$\equiv \text{APP}^{l+m}\{F, G_1, \cdots, G_l, H_1, \cdots, H_m\}$.

(iii) $\text{App}^m \circ \langle F, G_1, \cdots, G_m\rangle \xrightarrow{*} \text{APP}^m\{F, G_1, \cdots, G_m\}$.

Proof. The proof is straightforward and therefore is omitted. □

The auxiliary combinator Id^n behaves like the identity function for n-tuples. But Id^n is not a term of CCLMβ because it is an n-valued function. Rule (3) of CCLMβ is extended to the following derived rule.

PROPOSITION 3.2. *$F \circ \text{Id}^n \xrightarrow{*} F$ for every n-ary term F.*

Proof. The proof is by induction on the definition of F. □

The auxiliary combinators Λ^m and App^m are extensions of Λ and App, respectively. For $m \geqq 1$, the operator $\Lambda^m(—)$ means currying m times. Thus, for an $(m+n)$-argument function f, $\Lambda^m(f)$ is the n-argument function defined by

$$\Lambda^m(f) \equiv \underbrace{\Lambda(\Lambda(\cdots(\Lambda(f))\cdots))}_{m \text{ times}}.$$

Informally, in a λ-calculus-like notation, $\Lambda^m(f)$ means

$$\lambda\langle z_1, \cdots, z_n\rangle \cdot (\lambda x_1 \cdots \lambda x_m \cdot f(z_1, \cdots, z_n, x_1, \cdots, x_m)).$$

Likewise, for $m \geqq 1$, App^m receives $m+1$ arguments and applies the first argument to the other m arguments. It is informally represented by

$$\lambda\langle z, x_1, \cdots, x_m\rangle \cdot z x_1 \cdots x_m.$$

Rules (4) and (5) of CCLMβ have natural extensions for Λ^m and App^m.

PROPOSITION 3.3. *Let F be $(m+n)$-ary and let $G_1, \cdots, G_n$ be k-ary, where $m \geqq 1$. Then,*

$$\Lambda^m(F) \circ \langle G_1, \cdots, G_n\rangle \xrightarrow{*} \Lambda^m(F \circ \langle G_1 \circ P^{k,m}, \cdots, G_n \circ P^{k,m}, p_{k+1}^{k+m}, \cdots, p_{k+m}^{k+m}\rangle).$$

Proof. The proof is by induction on m. When $m = 1$ this is identical to rule (4).

$$\begin{aligned}
&\Lambda^{m+1}(F) \circ \langle G_1, \cdots, G_n\rangle \\
&\quad \to \Lambda(\Lambda^m(F) \circ \langle G_1 \circ P^{k,1}, \cdots, G_n \circ P^{k,1}, p_{k+1}^{k+1}\rangle) \quad \text{(by rule (4))} \\
&\quad \xrightarrow{*} \Lambda(\Lambda^m(F \circ \langle (G_1 \circ P^{k,1}) \circ P^{k+1,m}, \cdots, (G_n \circ P^{k,1}) \circ P^{k+1,m}, \\
&\qquad\qquad p_{k+1}^{k+1} \circ P^{k+1,m}, p_{k+2}^{k+1+m}, \cdots, p_{k+1+m}^{k+1+m}\rangle)) \quad \text{(by induction hypothesis)} \\
&\quad \xrightarrow{*} \Lambda^{m+1}(F \circ \langle G_1 \circ P^{k,m+1}, \cdots, G_n \circ P^{k,m+1}, p_{k+1}^{k+m+1}, \cdots, p_{k+m+1}^{k+m+1}\rangle). \qquad \square
\end{aligned}$$

LEMMA 3.4. *Let F be $(m+n)$-ary, and let $G_1, \cdots, G_m$ be n-ary, where $m \geqq 1$. Then,*

$$\mathrm{APP}^m \{\Lambda^m(F), G_1, \cdots, G_m\} \xrightarrow{*} F \circ \langle p_1^n, \cdots, p_n^n, G_1, \cdots, G_m\rangle.$$

Proof. The proof is by induction on m. When $m = 1$ this is rule (5).

$$\begin{aligned}
&\mathrm{APP}^{m+1} \{\Lambda^{m+1}(F), G_1, \cdots, G_m, G_{m+1}\} \\
&\quad \equiv \mathrm{App} \circ \langle \mathrm{APP}^m \{\Lambda^{m+1}(F), G_1, \cdots, G_m\}, G_{m+1}\rangle \\
&\quad \xrightarrow{*} \mathrm{App} \circ \langle \Lambda(F) \circ \langle p_1^n, \cdots, p_n^n, G_1, \cdots, G_m\rangle, G_{m+1}\rangle \quad \text{(by induction hypothesis)} \\
&\quad \to \mathrm{App} \circ \langle \Lambda(F \circ \langle p_1^n \circ P^{n,1}, \cdots, p_n^n \circ P^{n,1}, \\
&\qquad\qquad G_1 \circ P^{n,1}, \cdots, G_m \circ P^{n,1}, p_{n+1}^{n+1}\rangle), G_{m+1}\rangle \quad \text{(by rules (4), (7))} \\
&\quad \xrightarrow{*} \mathrm{App} \circ \langle \Lambda(F \circ \langle p_1^{n+1}, \cdots, p_n^{n+1}, G_1 \circ P^{n,1}, \cdots, G_m \circ P^{n,1}, p_{n+1}^{n+1}\rangle), G_{m+1}\rangle \\
&\quad \to (F \circ \langle p_1^{n+1}, \cdots, p_n^{n+1}, G_1 \circ P^{n,1}, \cdots, G_m \circ P^{n,1}, p_{n+1}^{n+1}\rangle) \\
&\qquad\qquad \circ \langle p_1^n, \cdots, p_n^n, G_{m+1}\rangle \quad \text{(by rule (5))}
\end{aligned}$$

$$\xrightarrow{*} F \circ \langle p_1^n, \cdots, p_n^n, G_1 \circ \mathrm{Id}^n, \cdots, G_m \circ \mathrm{Id}^n, G_{m+1} \rangle$$

$$\xrightarrow{*} F \circ \langle p_1^n, \cdots, p_n^n, G_1, \cdots, G_m, G_{m+1} \rangle \quad \text{(by Proposition 3.2).} \qquad \square$$

PROPOSITION 3.5. *Let F be $(m+n)$-ary, and let $G_1, \cdots, G_m$ be n-ary, where $m \geqq 1$. Then,*

$$\mathrm{App}^m \circ \langle \Lambda^m(F), G_1, \cdots, G_m \rangle \xrightarrow{*} F \circ \langle p_1^n, \cdots, p_n^n, G_1, \cdots, G_m \rangle.$$

Proof. The proof is immediate by combining Lemma 3.1(iii) and Lemma 3.4. $\square$

Example. We give an example of computation in CCLMβ with the auxiliary combinators. Let us use the same function plus $(x, y, z) = x + y + z$, and give two values 2 and 3 to $\lambda xyz \cdot \mathrm{plus}\,(x, y, z)$:

$$\mathrm{App}^2 \circ \langle \Lambda^3(\mathrm{plus} \circ \langle p_1^3, p_2^3, p_3^3 \rangle), 2^0, 3^0 \rangle$$

$$\xrightarrow{*} \Lambda(\mathrm{plus} \circ \langle p_1^3, p_2^3, p_3^3 \rangle) \circ \langle 2^0, 3^0 \rangle \quad \text{(by Proposition 3.5)}$$

$$\to \Lambda((\mathrm{plus} \circ \langle p_1^3, p_2^3, p_3^3 \rangle) \circ \langle 2^0 \circ \langle\,\rangle_1, 3^0 \circ \langle\,\rangle_1, p_1^1 \rangle) \quad \text{(by rule (4))}$$

$$\xrightarrow{*} \Lambda(\mathrm{plus} \circ \langle 2^1, 3^1, p_1^1 \rangle).$$

4. On models of CCLMβ. Before we examine the properties of CCLMβ as a rewriting system, we digress and make a brief discussion about models of CCLMβ as an equational system in Cartesian closed categories (CCC). Those who are interested only in the operational aspects of the system may skip this section.

Let $\mathbf{C}$ be a CCC. We say that an object U of $\mathbf{C}$ is *reflexive*, when there exists a pair of arrows $\phi : U \to U^U$ and $\psi : U^U \to U$ such that $\phi \circ \psi = id_{U^U}$. This means that U^U can be embedded into U. It is known that CCC's with reflexive object are essentially the same as λ-algebras (models of λ-calculus [2]). See, e.g., [11]. Similarly, CCC's with reflexive object characterize models of CCLMβ. We can naturally interpret terms of CCLMβ in $\mathbf{C}$ with reflexive object U. An n-ary term of CCLMβ is interpreted in the set $\mathbf{C}(U^n, U)$ of arrows from U^n to U. Here U^n denotes the product

$$1 \times \underbrace{U \times \cdots \times U}_{n \text{ times}}$$

where 1 is the terminal object of $\mathbf{C}$.

The interpretation of terms in $\mathbf{C}$ is the following. For each n-ary term, we define the arrow $[\![F]\!]$ from U^n to U in $\mathbf{C}$ as follows. Here we assume that for every nonspecial constant c of CCLMβ, $[\![c]\!]$ is already specified.

(1) $[\![p_i^n]\!] = \pi_{i+1}^{1,U,\cdots,U}$ (the $(i+1)$th projection from U^n to U).

(2) $[\![\mathrm{App}]\!] = ev^{U,U} \circ (\phi \times id_U)$, where $ev^{U,U} : U^U \times U \to U$ is the evaluation map.

(3) $[\![F \circ \langle G_1, \cdots, G_n \rangle]\!] = [\![F]\!] \circ \langle [\![G_1]\!], \cdots, [\![G_n]\!] \rangle$.

(4) $[\![\Lambda(F)]\!] = \psi \circ ([\![F]\!] \circ \pi_1^{U^n,U})^*$, where $h^* : U^n \to U^U$ is the transpose map of $h : U^n \times U \to U$.

Based on this interpretation we can prove that if $F = G$ in CCLMβ as an equational system, then $[\![F]\!] = [\![G]\!]$ in $\mathbf{C}$. That is, $\mathbf{C}$ is a model of CCLMβ.

5. Translations between CCLMβ and lambda calculus. In this section we define translation algorithms for both directions between CCLMβ and lambda calculus, and we establish the natural relationship between the terms in these two systems. The

lambda calculus we are concerned with is, more specifically, an extension of the type-free $\lambda\beta$-calculus, that we will denote by $\lambda\beta m$. The system $\lambda\beta m$ has constants with nonnegative integer called arity, just like CCLMβ. Since we intend to establish the relationship between CCLMβ and $\lambda\beta m$, we assume that there is given a one-to-one correspondence between the nonspecial constants of CCLMβ and the constants of $\lambda\beta m$.

DEFINITION. Terms of $\lambda\beta m$ are defined as follows:

(1) Variables are terms.

(2) If c is an m-ary constant and $N_1, \cdots, N_m$ are terms, then $c(N_1, \cdots, N_m)$ is a term.

(3) If M and N are terms, then MN is a term.

(4) If x is a variable and M is a term, then $\lambda x \cdot M$ is a term.

The rewriting rules of $\lambda\beta m$ are exactly the same as the ordinary $\lambda\beta$-calculus.

We provide notation for $\lambda\beta m$. Terms of $\lambda\beta m$ are denoted by M, N, M_1, etc., and variables are denoted by x, y, z, x_1, etc. When reduction $M \to N$ is derived from the rewriting rules of $\lambda\beta m$, we sometimes write $M \xrightarrow[\lambda\beta m]{} N$. For terms $M, N_1, \cdots, N_m$ of $\lambda\beta m$, we denote by $M[x_1, \cdots, x_m := N_1, \cdots, N_m]$ the term obtained from M by simultaneously substituting $N_1, \cdots, N_m$ for free occurrences of variables $x_1, \cdots, x_m$ in M. We write $M \equiv N$, when two terms M and N of $\lambda\beta m$ are the same except for bound variables.

Now we describe the translation algorithm from $\lambda\beta m$ to CCLMβ.

DEFINITION. Let $\langle z_1, \cdots, z_n\rangle$ be a sequence of distinct variables ($n \geqq 0$). For each term M of $\lambda\beta m$ whose free variables are contained in the set $\{z_1, \cdots, z_n\}$, we define the n-ary term $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]$ of CCLMβ as follows:

(1) $[\lambda\langle z_1, \cdots, z_n\rangle \cdot z_i] \equiv p_i^n$, $1 \leqq i \leqq n$.

(2) $[\lambda\langle z_1, \cdots, z_n\rangle \cdot c(N_1, \cdots, N_m)]$
$\equiv c \circ \langle[\lambda\langle z_1, \cdots, z_n\rangle \cdot N_1], \cdots, [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_m]\rangle$, where c is an m-ary constant.

(3) $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M_1 M_2] \equiv \mathrm{App} \circ \langle[\lambda\langle z_1, \cdots, z_n\rangle \cdot M_1], [\lambda\langle z_1, \cdots, z_n\rangle \cdot M_2]\rangle$.

(4) $[\lambda\langle z_1, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)] \equiv \Lambda([\lambda\langle z_1, \cdots, z_n, x'\rangle \cdot M_1[x := x']])$, where $x' \equiv x$ if x is not in $\{z_1, \cdots, z_n\}$, otherwise x' is a new variable.

In the following discussions, whenever we mention $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]$, we assume that the variables $z_1, \cdots, z_n$ are distinct and that all the free variables in M are contained in $\{z_1, \cdots, z_n\}$.

Next we give the translation algorithm from CCLMβ to $\lambda\beta m$.

DEFINITION. For each n-ary term F of CCLMβ and terms $N_1, \cdots, N_n$ of $\lambda\beta m$, we define the term $F^*[N_1, \cdots, N_n]$ of $\lambda\beta m$ as follows:

(1) $(p_i^n)^*[N_1, \cdots, N_n] \equiv N_i$, $1 \leqq i \leqq n$.

(2) $\mathrm{App}^*[N_1, N_2] \equiv N_1 N_2$.

(3) $c^*[N_1, \cdots, N_n] \equiv c(N_1, \cdots, N_n)$ for each n-ary nonspecial constant c.

(4) $(F \circ \langle G_1, \cdots, G_m\rangle)^*[N_1, \cdots, N_n]$
$\equiv F^*[G_1^*[N_1, \cdots, N_n], \cdots, G_m^*[N_1, \cdots, N_n]]$.

(5) $(\Lambda(F))^*[N_1, \cdots, N_n] \equiv \lambda x \cdot F^*[N_1, \cdots, N_n, x]$, where x is a variable not free in $N_1, \cdots, N_n$.

An n-ary term F of CCLMβ means an n-ary function. Thus F is intuitively represented by a $\lambda\beta m$ term M with free variables $x_1, \cdots, x_n$. In the above definition, $F^*[N_1, \cdots, N_n]$ means $M[x_1, \cdots, x_n := N_1, \cdots, N_n]$. Thus, this notation consists of two parts: translation of F into a term of $\lambda\beta m$ with n free variables, and substitution for the free variables. Actually, we can verify that

$$F^*[N_1, \cdots, N_n] \equiv (F^*[x_1, \cdots, x_n])[x_1, \cdots, x_n := N_1, \cdots, N_n].$$

Now we return to the former translation algorithm and give basic lemmas concerning it. To distinguish reductions in CCLMβ and $\lambda\beta m$, we write $F \xrightarrow[\text{CCLM}\beta]{} G$ when the reduction is derived in CCLMβ. In particular, when $F \xrightarrow[\text{CCLM}\beta]{} G$ is derived without rule (5) for App, we write $F \xrightarrow[\text{SUBM}]{} G$. Rule (5) corresponds to the β-rule of λ-calculus. The other rules have various good properties. For example, the relation $\xrightarrow[\text{SUBM}]{}$ is noetherian; namely, there is no infinite sequence $F_1, F_2, \cdots$ of terms such that $F_1 \xrightarrow[\text{SUBM}]{} F_2 \xrightarrow[\text{SUBM}]{} \cdots$. We refer to [8], which deals with a subsystem of CCL$\beta\eta$SP. See also [6]. Moreover, when we examine the properties of the above translation algorithms, it is convenient to separate rule (5) from the other rules. This is the reason for introducing SUBM as a subsystem of CCLMβ.

LEMMA 5.1.

(i) $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M] \equiv [\lambda\langle z'_1, \cdots, z'_n\rangle \cdot M[z_1, \cdots, z_n := z'_1, \cdots, z'_n]]$.

(ii) $[\lambda\langle x_1, \cdots, x_l, z_1, \cdots, z_n\rangle \cdot M] \circ \langle p_1^{l+m+n}, \cdots, p_l^{l+m+n}, p_{l+m+1}^{l+m+n}, \cdots, p_{l+m+n}^{l+m+n}\rangle$

$\xrightarrow[\text{SUBM}]{*} [\lambda\langle x_1, \cdots, x_l, y_1, \cdots, y_m, z_1, \cdots, z_n\rangle \cdot M]$.

(iii) $[\lambda\langle x_1, \cdots, x_m\rangle \cdot M] \circ \langle [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_1], \cdots, [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_m]\rangle$

$\xrightarrow[\text{SUBM}]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot M[x_1, \cdots, x_m := N_1, \cdots, N_m]]$.

Proof. The proof of (i) is easy and therefore is omitted.

(ii) The proof is by induction on the structure of M. We treat only the essential case: $M \equiv \lambda x \cdot M_1$. The other cases are straightforward. By (i) we can assume that x is not contained in $\{x_1, \cdots, x_l, y_1, \cdots, y_m, z_1, \cdots, z_n\}$:

$$[\lambda\langle x_1, \cdots, x_l, z_1, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)] \circ \langle p_1^{l+m+n}, \cdots, p_l^{l+m+n}, p_{l+m+1}^{l+m+n}, \cdots, p_{l+m+n}^{l+m+n}\rangle$$

$$\equiv \Lambda([\lambda\langle x_1, \cdots, x_l, z_1, \cdots, z_n, x\rangle \cdot M_1]) \circ \langle p_1^{l+m+n}, \cdots, p_l^{l+m+n}, p_{l+m+1}^{l+m+n}, \cdots, p_{l+m+n}^{l+m+n}\rangle$$

$$\xrightarrow[\text{SUBM}]{*} \Lambda([\lambda\langle x_1, \cdots, x_l, z_1, \cdots, z_n, x\rangle \cdot M_1] \circ \langle p_1^{l+m+n+1}, \cdots, p_l^{l+m+n+1}, p_{l+m+1}^{l+m+n+1}, \cdots, p_{l+m+n}^{l+m+n+1}, p_{l+m+n+1}^{l+m+n+1}\rangle)$$

$\xrightarrow[\text{SUBM}]{*} \Lambda([\lambda\langle x_1, \cdots, x_l, y_1, \cdots, y_m, z_1, \cdots, z_n, x\rangle \cdot M_1])$ (by induction hypothesis)

$\equiv [\lambda\langle x_1, \cdots, x_l, y_1, \cdots, y_m, z_1, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)]$.

(iii) The proof is by induction on the structure of M. When $M \equiv \lambda x \cdot M_1$,

$$[\lambda\langle x_1, \cdots, x_m\rangle \cdot (\lambda x \cdot M_1)] \circ \langle [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_1], \cdots, [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_m]\rangle$$

$$\xrightarrow[\text{SUBM}]{} \Lambda([\lambda\langle x_1, \cdots, x_m, x\rangle \cdot M_1] \circ \langle [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_1] \circ \langle p_1^{n+1}, \cdots, p_n^{n+1}\rangle,$$

$$\cdots, [\lambda\langle z_1, \cdots, z_n\rangle \cdot N_m] \circ \langle p_1^{n+1}, \cdots, p_n^{n+1}\rangle, p_{n+1}^{n+1}\rangle)$$

$$\xrightarrow[\text{SUBM}]{*} \Lambda([\lambda\langle x_1, \cdots, x_m, x\rangle \cdot M_1] \circ \langle [\lambda\langle z_1, \cdots, z_n, x\rangle \cdot N_1],$$

$$\cdots, [\lambda\langle z_1, \cdots, z_n, x\rangle \cdot N_m], [\lambda\langle z_1, \cdots, z_n, x\rangle \cdot x]\rangle) \quad \text{(by (ii))}$$

$$\xrightarrow[\text{SUBM}]{*} \Lambda([\lambda\langle z_1, \cdots, z_n, x\rangle \cdot M_1[x_1, \cdots, x_m := N_1, \cdots, N_m]])$$

(by induction hypothesis)

$$\equiv [\lambda\langle z_1, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)[x_1, \cdots, x_m := N_1, \cdots, N_m]].$$

The other cases are straightforward. □

6. Relationship between CCLMβ and $\lambda\beta m$. Now we are in a position to state the theorems that describe the relationship between the terms and reductions of the two systems CCLMβ and $\lambda\beta m$, in terms of the two translation algorithms of the previous section.

First, we show that the two translation algorithms preserve the reduction relations in CCLMβ and $\lambda\beta m$.

THEOREM 6.1. *If* $M \xrightarrow[\lambda\beta m]{*} N$, *then* $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M] \xrightarrow[\text{CCLM}\beta]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot N]$.

Proof. The proof is by induction on the definition of $M \xrightarrow[\lambda\beta m]{*} N$. We treat only the β-rule. The other rules are clear:

$$[\lambda\langle z_1, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)M_2]$$
$$\equiv \text{App} \circ \langle \Lambda([\lambda\langle z_1, \cdots, z_n, x\rangle \cdot M_1]), [\lambda\langle z_1, \cdots, z_n\rangle \cdot M_2]\rangle$$
$$\xrightarrow[\text{CCLM}\beta]{} [\lambda\langle z_1, \cdots, z_n, x\rangle \cdot M_1] \circ \langle p_1^n, \cdots, p_n^n, [\lambda\langle z_1, \cdots, z_n\rangle \cdot M_2]\rangle$$
$$\equiv [\lambda\langle z_1, \cdots, z_n, x\rangle \cdot M_1] \circ \langle [\lambda\langle z_1, \cdots, z_n\rangle \cdot z_1], \cdots,$$
$$[\lambda\langle z_1, \cdots, z_n\rangle \cdot z_n], [\lambda\langle z_1, \cdots, z_n\rangle \cdot M_2]\rangle$$
$$\xrightarrow[\text{SUBM}]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot M_1[x := M_2]] \quad \text{(by Lemma 5.1(iii))}. \qquad \square$$

THEOREM 6.2. *Let* F *and* G *be* n*-ary terms of* CCLMβ. *If* $F \xrightarrow[\text{CCLM}\beta]{*} G$ *then* $F^*[N_1, \cdots, N_n] \xrightarrow[\lambda\beta m]{*} G^*[N_1, \cdots, N_n]$. *In particular, if* $F \xrightarrow[\text{SUBM}]{*} G$ *then* $F^*[N_1, \cdots, N_n] \equiv G^*[N_1, \cdots, N_n]$.

Proof. The proof is by induction on the definition of $F \xrightarrow[\text{CCLM}\beta]{*} G$.

Case 1. $F \equiv \Lambda(H) \circ \langle H_1, \cdots, H_m\rangle \to G \equiv \Lambda(H \circ \langle H_1 \circ \langle p_1^{n+1}, \cdots, p_n^{n+1}\rangle, \cdots, H_m \circ \langle p_1^{n+1}, \cdots, p_n^{n+1}\rangle, p_{n+1}^{n+1}\rangle)$.

$$F^*[N_1, \cdots, N_n] \equiv \lambda x \cdot H^*[H_1^*[N_1, \cdots, N_n], \cdots, H_m^*[N_1, \cdots, N_n], x]$$
$$\equiv G^*[N_1, \cdots, N_n].$$

Case 2. $F \equiv \text{App} \circ \langle \Lambda(H_1), H_2\rangle \to G \equiv H_1 \circ \langle p_1^n, \cdots, p_n^n, H_2\rangle$.

$$F^*[N_1, \cdots, N_n] \equiv (\lambda x \cdot H_1^*[N_1, \cdots, N_n, x])(H_2^*[N_1, \cdots, N_n])$$
$$\xrightarrow[\lambda\beta m]{} H_1^*[N_1, \cdots, N_n, H_2^*[N_1, \cdots, N_n]]$$
$$\equiv G^*[N_1, \cdots, N_n].$$

The other cases are similar and omitted. $\square$

Next we examine the situation where a term of $\lambda\beta m$ is translated into CCLMβ and then the resulting term is translated back to $\lambda\beta m$.

THEOREM 6.3. $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^*[z_1, \cdots, z_n] \equiv M$.

Proof. The proof is by induction on the structure of M. When $M \equiv \lambda x \cdot M_1$,

$$[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^*[z_1, \cdots, z_n] \equiv (\Lambda([\lambda\langle z_1, \cdots, z_n, x\rangle \cdot M_1]))^*[z_1, \cdots, z_n]$$
$$\equiv \lambda x \cdot [\lambda\langle z_1, \cdots, z_n, x\rangle \cdot M_1]^*[z_1, \cdots, z_n, x]$$
$$\equiv \lambda x \cdot M_1 \quad \text{(by induction hypothesis)}.$$

The other cases are similar and omitted. $\square$

Using this theorem we can show the converse of Theorem 6.1.

COROLLARY 6.4. $M \xrightarrow[\lambda\beta m]{*} N$ *if and only if* $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M] \xrightarrow[\text{CCLM}\beta]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot N]$.

Proof. The only-if part is Theorem 6.1. For the if part, suppose the latter reduction. Then, by Theorem 6.2, we have

$$[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^*[z_1, \cdots, z_n] \xrightarrow[\lambda\beta m]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot N]^*[z_1, \cdots, z_n].$$

Therefore, by Theorem 6.3, $M \xrightarrow[\lambda\beta m]{*} N$. $\square$

7. Church–Rosser property. In this section we prove the Church–Rosser property of CCLMβ as an application of the results in the previous sections.

First, we examine the translated term $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]$ in CCLMβ. This term is of a special form. It contains no subterms of the following forms: $(F \circ \langle G_1, \cdots, G_m\rangle) \circ \langle H_1, \cdots, H_n\rangle$, $\Lambda(F) \circ \langle G_1, \cdots, G_m\rangle$, and $p_i^n \circ \langle F_1, \cdots, F_n\rangle$. But, it may contain subterms of the forms: $\mathrm{App} \circ \langle p_1^2, p_2^2\rangle$ and $c \circ \langle p_1^n, \cdots, p_n^n\rangle$, where c is an n-ary nonspecial constant. These two subterms are generated only by

$$[\lambda\langle z_1, z_2\rangle \cdot z_1 z_2] \equiv \mathrm{App} \circ \langle p_1^2, p_2^2\rangle$$

and

$$[\lambda\langle z_1, \cdots, z_n\rangle \cdot c(z_1, \cdots, z_n)] \equiv c \circ \langle p_1^n, \cdots, p_n^n\rangle,$$

respectively. If we replace subterms $\mathrm{App} \circ \langle p_1^2, p_2^2\rangle$ and $c \circ \langle p_1^n, \cdots, p_n^n\rangle$ by App and c, respectively, the resulting term F is in SUBM-normal form. Namely, there is no G such that $F \xrightarrow[\text{SUBM}]{} G$.

These observations bring us to the following definitions and theorems.

DEFINITION. For each term F of CCLMβ, we define F° as the term obtained from F by replacing all occurrences of subterms $\mathrm{App} \circ \langle p_1^2, p_2^2\rangle$ and $c \circ \langle p_1^n, \cdots, p_n^n\rangle$ by App and c, respectively, where c is an n-ary nonspecial constant.

Although we define F° for all F, our interest is in the special case where F is $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]$. All the lemmas and theorems in §§ 5 and 6 are still valid, even when we replace $[\lambda\langle\text{—}\rangle \cdot \text{—}]$ by $[\lambda\langle\text{—}\rangle \cdot \text{—}]^\circ$. In particular, we get the following theorem, an extended version of Corollary 6.4.

THEOREM 7.1. *$M \xrightarrow[\lambda\beta m]{*} N$ if and only if $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^\circ \xrightarrow[\text{CCLM}\beta]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot N]^\circ$.*

Using the translation algorithms between CCLMβ and $\lambda\beta m$, and the operation $(\text{—})^\circ$, we can define an algorithm that finds the SUBM-normal form of each term of CCLMβ.

DEFINITION. For each n-ary term F of CCLMβ, we define the term norm (F) by

$$\mathrm{norm}\,(F) \equiv [\lambda\langle z_1, \cdots, z_n\rangle \cdot F^*[z_1, \cdots, z_n]]^\circ.$$

THEOREM 7.2. $F \xrightarrow[\text{SUBM}]{*} \mathrm{norm}\,(F)$.

Proof. The proof is by induction on the structure of F.

Case 1. $F \equiv \mathrm{App}$.

$$\begin{aligned}\mathrm{norm}\,(\mathrm{App}) &\equiv [\lambda\langle z_1, z_2\rangle \cdot z_1 z_2]^\circ \\ &\equiv (\mathrm{App} \circ \langle p_1^2, p_2^2\rangle)^\circ \\ &\equiv \mathrm{App}.\end{aligned}$$

Case 2. $F \equiv H \circ \langle G_1, \cdots, G_m\rangle$.

$H \circ \langle G_1, \cdots, G_m\rangle$

$\xrightarrow[\text{SUBM}]{*} \mathrm{norm}\,(H) \circ \langle \mathrm{norm}\,(G_1), \cdots, \mathrm{norm}\,(G_m)\rangle$ (by induction hypothesis)

$\xrightarrow[\text{SUBM}]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot H^*[G_1^*[z_1, \cdots, z_n], \cdots, G_m^*[z_1, \cdots, z_n]]]^\circ$

(by the extended version of Lemma 5.1(iii))

$\equiv \mathrm{norm}\,(H \circ \langle G_1, \cdots, G_m\rangle)$.

Case 3. $F \equiv \Lambda(H)$. By induction hypothesis, we have

$$\Lambda(H) \xrightarrow[\text{SUBM}]{*} \Lambda(\mathrm{norm}\,(H)) \equiv \mathrm{norm}\,(\Lambda(H)).$$

The other cases are similar. □

By Theorem 7.2, $F \xrightarrow[\text{CCLM}\beta]{*} \text{norm}(F)$, and by definition, there is no G such that $\text{norm}(F) \xrightarrow[\text{SUBM}]{} G$. Moreover, if $F \xrightarrow[\text{SUBM}]{*} H$ then $H \xrightarrow[\text{SUBM}]{*} \text{norm}(H) \equiv \text{norm}(F)$ by Theorems 7.2 and 6.2. These mean that norm (F) is the unique SUBM-normal form of F.

Finally we establish the Church-Rosser property of CCLMβ.

THEOREM 7.3. *If* $F \xrightarrow[\text{CCLM}\beta]{*} G_1$ *and* $F \xrightarrow[\text{CCLM}\beta]{*} G_2$, *then there exists* H *such that* $G_1 \xrightarrow[\text{CCLM}\beta]{*} H$ *and* $G_2 \xrightarrow[\text{CCLM}\beta]{*} H$.

Proof. First note that $\lambda\beta m$ satisfies the Church-Rosser property as well as the ordinary $\lambda\beta$-calculus. Suppose $F \xrightarrow[\text{CCLM}\beta]{*} G_1$ and $F \xrightarrow[\text{CCLM}\beta]{*} G_2$. Assume that F is n-ary; so G_1 and G_2 are also n-ary. By Theorem 6.2 we have

$$F^*[z_1, \cdots, z_n] \xrightarrow[\lambda\beta m]{*} G_1^*[z_1, \cdots, z_n] \quad \text{and} \quad F^*[z_1, \cdots, z_n] \xrightarrow[\lambda\beta m]{*} G_2^*[z_1, \cdots, z_n].$$

By the Church-Rosser theorem of $\lambda\beta m$, there exists M such that

$$G_1^*[z_1, \cdots, z_n] \xrightarrow[\lambda\beta m]{*} M \quad \text{and} \quad G_2^*[z_1, \cdots, z_n] \xrightarrow[\lambda\beta m]{*} M.$$

By Theorem 7.1,

$$\text{norm}(G_1) \xrightarrow[\text{CCLM}\beta]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^\circ$$

and

$$\text{norm}(G_2) \xrightarrow[\text{CCLM}\beta]{*} [\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^\circ.$$

By Theorem 7.2,

$$G_1 \xrightarrow[\text{SUBM}]{*} \text{norm}(G_1) \quad \text{and} \quad G_2 \xrightarrow[\text{SUBM}]{*} \text{norm}(G_2).$$

Therefore, if we take $[\lambda\langle z_1, \cdots, z_n\rangle \cdot M]^\circ$ for H, we get

$$G_1 \xrightarrow[\text{CCLM}\beta]{*} H \quad \text{and} \quad G_2 \xrightarrow[\text{CCLM}\beta]{*} H. \qquad \square$$

8. The system CCLβ for original categorical combinators. In this section, we return to the original system CCLβ introduced by Curien, and establish the relationship between CCLβ and λ-calculus with product in a similar method of the previous sections. The system CCLβ does not satisfy the Church-Rosser property, but there are various subsets D on which CCLβ satisfies the Church-Rosser property [20], [9]. These facts have been proved directly without help of the Church-Rosser theorem of λ-calculus. Here, we show that a similar result with regard to the Church-Rosser property for CCLβ is immediately derived from the relationship between CCLβ and the λ-calculus.

First we present CCLβ following [4].

DEFINITION. We assume that constant symbols are specified in CCLβ. In particular, they always include special constants: Fst, Snd, Id, and App. Terms of CCLβ are defined as follows:

(1) Every constant is a term.

(2) If F and G are terms, then $F \circ G$ and $\langle F, G\rangle$ are terms.

(3) If F is a term, then $\Lambda(F)$ is a term.

DEFINITION. We define the binary relation $\rightarrow$ among the terms of CCLβ by the following rules:

(1) $(F \circ G) \circ H \rightarrow F \circ (G \circ H)$.

(2) $\text{Fst} \circ \langle F, G\rangle \rightarrow F$.

(3) $\text{Snd} \circ \langle F, G\rangle \rightarrow G$.

(4) $\text{Id} \circ F \rightarrow F$.

(5) $F \circ \mathrm{Id} \to F$.

(6) $\Lambda(F) \circ G \to \Lambda(F \circ \langle G \circ \mathrm{Fst}, \mathrm{Snd}\rangle)$.

(7) $\mathrm{App} \circ \langle \Lambda(F), G\rangle \to F \circ \langle \mathrm{Id}, G\rangle$.

(8) If $F \to G$, then $H \circ F \to H \circ G$, $F \circ H \to G \circ H$, $\langle F, H\rangle \to \langle G, H\rangle$, $\langle H, F\rangle \to \langle H, G\rangle$, and $\Lambda(F) \to \Lambda(G)$.

When $F \to G$ is derived from the rules of CCLβ, we write $F \xrightarrow[\mathrm{CCL}\beta]{} G$. In particular, if $F \to G$ is derived without rule (7), we sometimes write $F \xrightarrow[\mathrm{SUB}]{} G$.

We provide notation to represent various terms of CCLβ.

Notation.

(i) $F_1 \circ F_2 \circ \cdots \circ F_{n-1} \circ F_n$ is an abbreviation for $F_1 \circ (F_2 \circ \cdots \circ (F_{n-1} \circ F_n) \cdots)$.

(ii) $\langle F_1, F_2, F_3, \cdots, F_n\rangle$ is an abbreviation for $\langle \cdots \langle\langle F_1, F_2\rangle, F_3\rangle, \cdots, F_n\rangle$. When $n = 1$, we define $\langle F_1\rangle \equiv F_1$.

(iii) For $n \geqq 1$, define the term Fst^n of CCLβ by

$$\mathrm{Fst}^n \equiv \underbrace{\mathrm{Fst} \circ \mathrm{Fst} \circ \cdots \mathrm{Fst}}_{n \text{ times}}.$$

In particular, when $n = 0$, we define $F \circ \mathrm{Fst}^0 \equiv F$ for each term F.

(iv) For $n \geqq 0$ and $0 \leqq i \leqq n$, define the term π_i^n of CCLβ as follows:

$$\pi_0^0 \equiv \mathrm{Id},$$

$$\pi_0^n \equiv \mathrm{Fst}^n \quad \text{if } n \geqq 1,$$

$$\pi_i^n \equiv \mathrm{Snd} \circ \mathrm{Fst}^{n-i} \quad \text{if } 1 \leqq i \leqq n.$$

(v) For $n \geqq 0$ and a term H, define the term $\Pi_n(H)$ of CCLβ by $\Pi_0(H) \equiv H$ and $\Pi_{n+1}(H) \equiv \langle \Pi_n(H) \circ \mathrm{Fst}, \mathrm{Snd}\rangle$.

Next we provide a formal system for λ-calculus corresponding to CCLβ. Extending the ordinary $\lambda\beta$-calculus, we define a rewriting system called $\lambda\beta c$.

DEFINITION. We assume that constant symbols are specified in $\lambda\beta c$. In particular, they always include special constants: fst and snd. Terms of $\lambda\beta c$ are defined as follows:

(1) Every variable is a term.

(2) If x is a variable and M is a term, then $\lambda x \cdot M$ is a term.

(3) If M and N are terms, then MN and $\langle M, N\rangle$ are terms.

(4) If c is a constant symbol and M is a term, then $c(M)$ is a term.

The rewriting rules for $\lambda\beta c$ are those for the ordinary $\lambda\beta$-calculus, together with the following two:

$$\mathrm{fst}\,(\langle M_1, M_2\rangle) \to M_1, \qquad \mathrm{snd}\,(\langle M_1, M_2\rangle) \to M_2.$$

When $M \to N$ is derived in $\lambda\beta c$, we write $M \xrightarrow[\lambda\beta c]{} N$.

In a similar way to those for CCLMβ and $\lambda\beta m$ of § 5, we define a pair of translation algorithms between CCLβ and $\lambda\beta c$. We assume, as before, that there is given a one-to-one correspondence between the nonspecial constants of CCLβ and $\lambda\beta c$.

DEFINITION. Let $\langle z_0, \cdots, z_n\rangle$ be a *nonempty* sequence of distinct variables. For each term M of $\lambda\beta c$ whose free variables are contained in the set $\{z_0, \cdots, z_n\}$, we define the term $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M]$ of CCLβ as follows:

(1) $[\lambda\langle z_0, \cdots, z_n\rangle \cdot z_i] \equiv \pi_i^n$, $0 \leqq i \leqq n$.

(2) $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M_1 M_2] \equiv \mathrm{App} \circ \langle [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_1], [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_2]\rangle$.

(3) $[\lambda\langle z_0, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)] \equiv \Lambda([\lambda\langle z_0, \cdots, z_n, x'\rangle \cdot M_1[x := x']])$, where $x' \equiv x$ if x is not in $\{z_0, \cdots, z_n\}$, otherwise x' is a new variable.

(4) $[\lambda\langle z_0, \cdots, z_n\rangle \cdot c(M_1)] \equiv c \circ [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_1]$, where c is a nonspecial constant.

(5) $[\lambda\langle z_0, \cdots, z_n\rangle \cdot \mathrm{fst}\,(M_1)] \equiv \mathrm{Fst} \circ [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_1]$,
$[\lambda\langle z_0, \cdots, z_n\rangle \cdot \mathrm{snd}\,(M_1)] \equiv \mathrm{Snd} \circ [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_1]$.

The above translation algorithm is essentially the same as $[\![M]\!]_\Delta$ defined by Koymans [11, p. 314], and also as $M_{DB(y_0,\cdots,y_n)}$ defined by Curien [4, p. 201]. More precisely, $[\![M]\!]_{\langle x_1,\cdots,x_m\rangle}$ of the former corresponds to $[\lambda\langle z, x_1, \cdots, x_m\rangle \cdot M]$, where z is a variable distinct from $x_1, \cdots, x_m$. But Koymans treats the translation more semantically. Similarly, $M_{DB(y_0,\cdots,y_n)}$ coincides with $[\lambda\langle z, y_n, \cdots, y_0\rangle \cdot M]$. Note that the order of variables $y_0, \cdots, y_n$ is reversed, because $M_{DB(y_0,\cdots,y_n)}$ is based on De Bruijn's notation. The significant difference of our translation algorithm from $[\![M]\!]_\Delta$ and $M_{DB(y_0,\cdots,y_n)}$ is shown by the following examples:

$$[\lambda\langle z\rangle \cdot z] \equiv \mathrm{Id}, \qquad [\![z]\!]_{\langle z\rangle} \equiv z_{DB(z)} \equiv \mathrm{Snd},$$

$$[\lambda\langle z\rangle \cdot (\lambda x \cdot z)] \equiv \Lambda(\mathrm{Fst}), \qquad [\![\lambda x \cdot z]\!]_{\langle z\rangle} \equiv (\lambda x \cdot z)_{DB(z)} \equiv \Lambda(\mathrm{Snd} \circ \mathrm{Fst}).$$

Neither $[\![M]\!]_\Delta$ nor $M_{DB(y_0,\cdots,y_n)}$ can express Id, $\Lambda(\mathrm{Fst})$, and so on.

Conversely, we define a translation algorithm from CCLβ to $\lambda\beta c$.

DEFINITION. For each pair of a term F of CCLβ and a term N of $\lambda\beta c$, we define the term $F^*[N]$ of $\lambda\beta c$ as follows:

(1) $\mathrm{Id}^*[N] \equiv N$.

(2) $\mathrm{Fst}^*[N] \equiv \begin{cases} N_1 & \text{if } N \equiv \langle N_1, N_2\rangle, \\ \mathrm{fst}\,(N) & \text{otherwise.} \end{cases}$

(3) $\mathrm{Snd}^*[N] \equiv \begin{cases} N_2 & \text{if } N \equiv \langle N_1, N_2\rangle, \\ \mathrm{snd}\,(N) & \text{otherwise.} \end{cases}$

(4) $\mathrm{App}^*[N] \equiv \begin{cases} N_1 N_2 & \text{if } N \equiv \langle N_1, N_2\rangle, \\ \mathrm{fst}\,(N)\ \mathrm{snd}\,(N) & \text{otherwise.} \end{cases}$

(5) $c^*[N] \equiv c(N)$, where c is a nonspecial constant.

(6) $\langle F_1, F_2\rangle^*[N] \equiv \langle F_1^*[N], F_2^*[N]\rangle$.

(7) $(\Lambda(F_1))^*[N] \equiv \lambda x \cdot F_1^*[\langle N, x\rangle]$, where x is a variable not free in N.

(8) $(F_1 \circ F_2)^*[N] \equiv F_1^*[F_2^*[N]]$.

The following are basic properties of $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M]$, similar to Lemma 5.1.

LEMMA 8.1.

(i) $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M] \equiv [\lambda\langle z_0', \cdots, z_n'\rangle \cdot M[z_0, \cdots, z_n := z_0', \cdots, z_n']]$.

(ii) $[\lambda\langle x_0, x_1, \cdots, x_l, z_1, \cdots, z_n\rangle \cdot M] \circ \Pi_n(\mathrm{Fst}^m)$
$\xrightarrow[\mathrm{SUB}]{*} [\lambda\langle x_0, x_1, \cdots, x_l, y_1, \cdots, y_m, z_1, \cdots, z_n\rangle \cdot M]$.

(iii) $[\lambda\langle x_0, \cdots, x_m\rangle \cdot M] \circ \langle [\lambda\langle z_0, \cdots, z_n\rangle \cdot N_0], \cdots, [\lambda\langle z_0, \cdots, z_n\rangle \cdot N_m]\rangle$
$\xrightarrow[\mathrm{SUB}]{*} [\lambda\langle z_0, \cdots, z_n\rangle \cdot M[x_0, \cdots, x_m := N_0, \cdots, N_m]]$.

(iv) $[\lambda\langle z_0, \cdots, z_n, x_1, \cdots, x_m\rangle \cdot M]$
$\equiv [\lambda\langle z, x_1, \cdots, x_m\rangle \cdot M[z_0, \cdots, z_n := (\pi_0^n)^*[z], \cdots, (\pi_n^n)^*[z]]]$.

Proof. The lemma is easily proved by induction on the structure of M. □

Now we can present equivalence theorems between CCLβ and $\lambda\beta c$, similar to the results between CCLMβ and $\lambda\beta m$.

THEOREM 8.2.

(i) *If* $M \xrightarrow[\lambda\beta c]{*} N$, *then* $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M] \xrightarrow[\text{CCL}\beta]{*} [\lambda\langle z_0, \cdots, z_n\rangle \cdot N]$.

(ii) *If* $F \xrightarrow[\text{CCL}\beta]{*} G$, *then* $F^*[N] \xrightarrow[\lambda\beta c]{*} G^*[N]$. *In particular, if* $F \xrightarrow[\text{SUB}]{*} G$ *then* $F^*[N] \equiv G^*[N]$.

Proof. (i) The proof is by induction on the definition of $M \xrightarrow[\lambda\beta c]{*} N$. We treat only the β-rule. The other rules are clear:

$$
\begin{aligned}
&[\lambda\langle z_0, \cdots, z_n\rangle \cdot (\lambda x \cdot M_1)M_2] \\
&\quad \equiv \text{App} \circ \langle \Lambda([\lambda\langle z_0, \cdots, z_n, x\rangle \cdot M_1]), [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_2]\rangle \\
&\quad \xrightarrow[\text{CCL}\beta]{} [\lambda\langle z_0, \cdots, z_n, x\rangle \cdot M_1] \circ \langle \text{Id}, [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_2]\rangle \\
&\quad \equiv [\lambda\langle z, x\rangle \cdot M_1[z_0, \cdots, z_n := (\pi_0^n)^*[z], \cdots, (\pi_n^n)^*[z]]] \\
&\qquad\quad \circ \langle [\lambda\langle z\rangle \cdot z], [\lambda\langle z\rangle \cdot M_2[z_0, \cdots, z_n := (\pi_0^n)^*[z], \cdots, (\pi_n^n)^*[z]]]\rangle \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Lemma 8.1(iv))} \\
&\quad \xrightarrow[\text{SUB}]{*} [\lambda\langle z\rangle \cdot (M_1[x := M_2])[z_0, \cdots, z_n := (\pi_0^n)^*[z], \cdots, (\pi_n^n)^*[z]]] \\
&\qquad\qquad\qquad\qquad\qquad\qquad\qquad\qquad \text{(by Lemma 8.1(iii))} \\
&\quad \equiv [\lambda\langle z_0, \cdots, z_n\rangle \cdot M_1[x := M_2]] \quad \text{(by Lemma 8.1(iv))}.
\end{aligned}
$$

(ii) The proof is by induction on the definition of $F \xrightarrow[\text{CCL}\beta]{*} G$. When $F \equiv \text{App} \circ \langle \Lambda(F_1), F_2\rangle$ and $G \equiv F_1 \circ \langle \text{Id}, F_2\rangle$,

$$
\begin{aligned}
(\text{App} \circ \langle \Lambda(F_1), F_2\rangle)^*[N] &\equiv (\lambda x \cdot F_1^*[\langle N, x\rangle])(F_2^*[N]) \\
&\xrightarrow[\lambda\beta c]{} (F_1^*[\langle N, x\rangle])[x := F_2^*[N]] \\
&\xrightarrow[\lambda\beta c]{*} F_1^*[\langle N, F_2^*[N]\rangle] \\
&\equiv (F_1 \circ \langle \text{Id}, F_2\rangle)^*[N].
\end{aligned}
$$

Here note that, in general, $(G^*[M])[y := M'] \xrightarrow[\lambda\beta c]{*} G^*[M[y := M']]$, which is easily proved by induction on the structure of G.

The other rules are similar and omitted. □

Theorem 6.3 is somewhat complex in the case of CCLβ and $\lambda\beta c$. As an abbreviation of terms $\langle \cdots \langle\langle N_1, N_2\rangle, N_3\rangle, \cdots, N_m\rangle$ of $\lambda\beta c$, we sometimes write $\langle N_1, N_2, N_3, \cdots, N_m\rangle$. When $m = 1$ we define $\langle N_1\rangle \equiv N_1$. It is certain that

$$[\lambda\langle z_0, \cdots, z_n\rangle \cdot M]^*[\langle z_0, \cdots, z_n\rangle] \xrightarrow[\lambda\beta c]{*} M.$$

But, unfortunately, $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M]^*[\langle z_0, \cdots, z_n\rangle]$ and M are not generally identical. For example,

$$[\lambda\langle z\rangle \cdot \text{fst}\,(\langle z, z\rangle)]^*[z] \equiv z.$$

This is due to the following two different origins of the special constants fst and snd of $\lambda\beta c$ appearing in $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M]^*[\langle z_0, \cdots, z_n\rangle]$: (1) originally contained in M, and (2) newly introduced.

We can overcome this difficulty of distinguishing these two kinds of fst and snd by extending CCLβ and $\lambda\beta c$. We define the extended CCLβ by adding two constants F$\tilde{\text{s}}$t and S$\tilde{\text{n}}$d, and two rewriting rules: F$\tilde{\text{s}}$t $\circ \langle F, G\rangle \to F$ and S$\tilde{\text{n}}$d $\circ \langle F, G\rangle \to G$. Similarly, the extended $\lambda\beta c$ is defined by adding two constants f$\tilde{\text{s}}$t and s$\tilde{\text{n}}$d, and two rewriting rules: f$\tilde{\text{s}}$t $(\langle M, N\rangle) \to M$ and s$\tilde{\text{n}}$d $(\langle M, N\rangle) \to N$. The translation algorithms between

the extended CCLβ and $\lambda\beta c$ are defined so that F$\tilde{\text{s}}$t and S$\tilde{\text{n}}$d correspond to f$\tilde{\text{s}}$t and s$\tilde{\text{n}}$d, respectively. The difference between Fst and F$\tilde{\text{s}}$t affects only the translation algorithm $(\text{—})^*[\text{—}]$. For example, $\text{Fst}^*[\langle N_1, N_2\rangle] \equiv N_1$ but $\tilde{\text{Fst}}^*[\langle N_1, N_2\rangle] \equiv \tilde{\text{f}}\text{st}(\langle N_1, N_2\rangle)$. For each term M of $\lambda\beta c$, we define $\tilde{M}$ as the term obtained from M by replacing fst and snd by f$\tilde{\text{s}}$t and s$\tilde{\text{n}}$d, respectively. Then, we get the following theorem.

THEOREM 8.3. $[\lambda\langle z_0, \cdots, z_n\rangle \cdot \tilde{M}]^*[\langle z_0, \cdots, z_n\rangle] \equiv \tilde{M}$.

Proof. The proof is similar to that of Theorem 6.3. □

From Theorems 8.2 and 8.3, we get the following corollary.

COROLLARY 8.4. $M \xrightarrow[\lambda\beta c]{*} N$ *if and only if* $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M] \xrightarrow[\text{CCL}\beta]{*} [\lambda\langle z_0, \cdots, z_n\rangle \cdot N]$.

Proof. The proof is similar to that of Corollary 6.4. Note that Lemma 8.1 and Theorem 8.2 still hold for the extended CCLβ and $\lambda\beta c$. □

Next we examine the Church-Rosser property of CCLβ. In § 7, we defined $(\text{—})^\circ$ and norm (—) on terms of CCLMβ. Similar operations can be defined for CCLβ.

DEFINITION. For each term F of CCLβ, we define F° as the term obtained from F by replacing every subterm $c \circ \text{Id}$ in F by c, where c is a constant.

DEFINITION. For each term F of CCLβ, we define the term norm (F) of CCLβ by

$$\text{norm}(F) \equiv [\lambda\langle z\rangle \cdot F^*[z]]^\circ.$$

For a term M of $\lambda\beta c$, each subterm $c \circ \text{Id}$ appearing in $[\lambda\langle z_0, \cdots, z_n\rangle \cdot M]$ is generated only by $[\lambda\langle z_0\rangle \cdot c(z_0)] \equiv c \circ \text{Id}$. Similarly to the case of CCLMβ, Lemma 8.1, Theorems 8.2, 8.3, and Corollary 8.4 still hold, even when we replace $[\lambda\langle\text{—}\rangle \cdot \text{—}]$ by $[\lambda\langle\text{—}\rangle \cdot \text{—}]^\circ$.

Theorem 7.2 is troublesome. Unfortunately, the reduction $F \xrightarrow[\text{SUB}]{*} \text{norm}(F)$ does not generally hold in CCLβ and $\lambda\beta c$. For example,

$$\text{norm}(\Lambda(\text{Id})) \equiv \Lambda(\langle \text{Fst}, \text{Snd}\rangle).$$

We can only prove that $F \circ \text{Id} \xrightarrow[\text{SUB}]{*} \text{norm}(F)$ if App in F appears only in the form $\text{App} \circ \langle F_1, F_2\rangle$. However, this cannot be used to prove the Church-Rosser property of CCLβ, and actually CCLβ does not have the Church-Rosser property. The following theorem shows that CCLβ satisfies the Church-Rosser property if we restrict terms to a certain subset.

THEOREM 8.5. *Let F be a term that satisfies the condition: $F' \xrightarrow[\text{CCL}\beta]{*} \text{norm}(F')$ for any F' such that $F \xrightarrow[\text{CCL}\beta]{*} F'$. If $F \xrightarrow[\text{CCL}\beta]{*} G_1$ and $F \xrightarrow[\text{CCL}\beta]{*} G_2$, then there exists H such that $G_1 \xrightarrow[\text{CCL}\beta]{*} H$ and $G_2 \xrightarrow[\text{CCL}\beta]{*} H$.*

Proof. The proof is similar to that of Theorem 7.3. □

This theorem suggests that norm (F) is the key to examining the Church-Rosser property of CCLβ. Actually, norm (F) can be defined directly, and the Church-Rosser theorem on restricted terms for CCLβ is proved without using the theorem for λ-calculus [20]. In [20], various sets of terms that satisfy the condition of Theorem 8.5 are concretely defined. We shall present a set of such terms in § 9.

9. Embedding of CCLMβ into CCLβ. We show that CCLMβ is characterized as a subsystem of CCLβ. First we examine the relations between $\lambda\beta m$ and $\lambda\beta c$. Then, using the results in the previous sections, we translate these relations to those between CCLMβ and CCLβ. In this section, we treat four systems $\lambda\beta m$, $\lambda\beta c$, CCLMβ, and CCLβ, and assume that nonspecial constants are given in common with these systems.

To start with, we define a pair of translation rules single (—) and multi (—) between $\lambda\beta m$ and $\lambda\beta c$, and show that $\lambda\beta m$ is embedded into $\lambda\beta c$. Here we arbitrarily choose a closed term $\#$ of $\lambda\beta c$ that satisfies the following conditions and fix it such as, for example, $\# \equiv \lambda x \cdot x$:

(1) $\#$ is in normal form,

(2) $\#$ contains no constants, and

(3) $\#$ does not contain operator $\langle(—), (—)\rangle$.

DEFINITION. For each term M of $\lambda\beta m$, we define the term single (M) of $\lambda\beta c$ as follows:

(1) single $(x) \equiv x$,

(2) single $(M_1 M_2) \equiv$ single (M_1) single (M_2),

(3) single $(\lambda x \cdot M_1) \equiv \lambda x \cdot$ single (M_1),

(4) single $(c(N_1, \cdots, N_m)) \equiv c(\langle \#,$ single $(N_1), \cdots,$ single $(N_m)\rangle)$, where c is an n-ary nonspecial constant.

The resulting term by the translation single (—) has a special shape, that is characterized by the following definition.

DEFINITION. A term M of $\lambda\beta c$ is said to be *stable* when M satisfies the following conditions:

(1) M does not contain fst nor snd.

(2) Every constant c in M appears only in the form $c(\langle \#, N_1, \cdots, N_m\rangle)$.

(3) The operator $\langle(—), (—)\rangle$ appears only in the form of (2).

Note that single (M) is always stable for every term M of $\lambda\beta m$.

DEFINITION. For every stable term M of $\lambda\beta c$, we define the term multi (M) of $\lambda\beta m$ as follows:

(1) multi $(x) \equiv x$,

(2) multi $(M_1 M_2) \equiv$ multi (M_1) multi (M_2),

(3) multi $(\lambda x \cdot M_1) \equiv \lambda x \cdot$ multi (M_1),

(4) multi $(c(\langle \#, N_1, \cdots, N_m\rangle)) \equiv c($multi $(N_1), \cdots,$ multi $(N_m))$, where c is a nonspecial constant.

The following are basic properties of the translation rules single (—) and multi (—).

LEMMA 9.1.

(i) single (multi $(M)) \equiv M$ *for every stable term* M *of* $\lambda\beta c$.

(ii) multi (single $(N)) \equiv N$ *for every term* N *of* $\lambda\beta m$.

(iii) *Let* $M \xrightarrow[\lambda\beta c]{*} N$. *If* M *is stable then so is* N.

(iv) *Let* M *and* N *be stable terms of* $\lambda\beta c$. *Then,* $M \xrightarrow[\lambda\beta c]{*} N$ *if and only if* multi $(M) \xrightarrow[\lambda\beta m]{*}$ multi (N).

(v) $M \xrightarrow[\lambda\beta m]{*} N$ *if and only if* single $(M) \xrightarrow[\lambda\beta c]{*}$ single (N).

Proof. The proof is simple and therefore is omitted. □

Next, using the above translation rules between $\lambda\beta m$ and $\lambda\beta c$, we define a pair of translation rules between CCLMβ and CCLβ.

DEFINITION. For each term F of CCLMβ, we define the term single (F) of CCLβ by

$$\text{single } (F) \equiv [\lambda\langle z_0, z_1, \cdots, z_n\rangle \cdot \text{single } (F^*[z_1, \cdots, z_n])]^\circ.$$

The translation rule for the inverse direction from CCLβ to CCLMβ is defined on a restricted set of terms of CCLβ. The following definition characterizes terms of CCLβ that correspond to stable terms of $\lambda\beta c$.

DEFINITION. Let n be a nonnegative integer, and let $z_0, \cdots, z_n$ be distinct variables. A term F of CCLβ is said to be *n-stable* when F satisfies the following conditions:

(1) $F^*[\langle z_0, \cdots, z_n\rangle]$ is stable in $\lambda\beta c$, and

(2) $F^*[\langle z_0, \cdots, z_n\rangle]$ does not contain z_0.

DEFINITION. For each n-stable term F of CCLβ, we define the n-ary term $\mathrm{multi}_n(F)$ of CCLMβ by

$$\mathrm{multi}_n(F) \equiv [\lambda\langle z_1, \cdots, z_n\rangle \cdot \mathrm{multi}(F^*[\langle z_0, z_1, \cdots, z_n\rangle])]^\circ.$$

Note that $F^*[\langle z_0, z_1, \cdots, z_n\rangle]$ does not contain z_0 so that the right-hand side expression is well defined.

The following theorem shows that $\mathrm{multi}_n(—)$ and $\mathrm{single}(—)$ preserve reductions in CCLMβ and CCLβ.

THEOREM 9.2.

(i) *Let F be an n-stable term of* CCLβ. If $F \xrightarrow[\mathrm{CCL}\beta]{*} G$, *then G is n-stable and* $\mathrm{multi}_n(F) \xrightarrow[\mathrm{CCLM}\beta]{*} \mathrm{multi}_n(G)$.

(ii) *If* $F \xrightarrow[\mathrm{CCLM}\beta]{*} G$, *then* $\mathrm{single}(F) \xrightarrow[\mathrm{CCL}\beta]{*} \mathrm{single}(G)$.

Proof. (i) Suppose that F is n-stable and that $F \xrightarrow[\mathrm{CCL}\beta]{*} G$. Then, by Theorem 8.2(ii),

$$F^*[\langle z_0, z_1, \cdots, z_n\rangle] \xrightarrow[\lambda\beta c]{*} G^*[\langle z_0, z_1, \cdots, z_n\rangle].$$

So, G is n-stable, too:

$F^*[\langle z_0, z_1, \cdots, z_n\rangle] \xrightarrow[\lambda\beta c]{*} G^*[\langle z_0, z_1, \cdots, z_n\rangle]$

$\Leftrightarrow \mathrm{multi}(F^*[\langle z_0, z_1, \cdots, z_n\rangle]) \xrightarrow[\lambda\beta m]{*} \mathrm{multi}(G^*[\langle z_0, z_1, \cdots, z_n\rangle])$ (by Lemma 9.1(iv))

$\Leftrightarrow \mathrm{multi}_n(F) \xrightarrow[\mathrm{CCLM}\beta]{*} \mathrm{multi}_n(G)$ (by Theorem 7.1).

(ii) $F \xrightarrow[\mathrm{CCLM}\beta]{*} G$

$\Rightarrow F^*[z_1, \cdots, z_n] \xrightarrow[\lambda\beta m]{*} G^*[z_1, \cdots, z_n]$ (by Theorem 6.2)

$\Leftrightarrow \mathrm{single}(F^*[z_1, \cdots, z_n]) \xrightarrow[\lambda\beta c]{*} \mathrm{single}(G^*[z_1, \cdots, z_n])$ (by Lemma 9.1(v))

$\Leftrightarrow \mathrm{single}(F) \xrightarrow[\mathrm{CCL}\beta]{*} \mathrm{single}(G)$ (by Corollary 8.4). □

Now we further examine the properties of n-stable terms and show that $F \xrightarrow[\mathrm{SUB}]{*} \mathrm{norm}(F)$ for any n-stable term F of CCLβ. Therefore, n-stable terms satisfy the condition of Theorem 8.5, and CCLβ satisfies the Church-Rosser property on n-stable terms. For the proof, we use a translation $\mathrm{norm}^*(—)$ on terms of CCLβ, which is introduced in [20]. Moreover, $\mathrm{norm}^*(—)$ coincides with $\mathscr{E}(—)$ defined in [9].

DEFINITION. For each term F of CCLβ we define the term $\mathrm{norm}^*(F)$ as follows:

(1) $\mathrm{norm}^*((F_1 \circ F_2) \circ H) \equiv \mathrm{norm}^*(F_1 \circ (F_2 \circ H))$.

(2) $\mathrm{norm}^*(\Lambda(F_1) \circ H) \equiv \Lambda(\mathrm{norm}^*(F_1 \circ \langle H \circ \mathrm{Fst}, \mathrm{Snd}\rangle))$.

(3) $\mathrm{norm}^*(\langle F_1, F_2\rangle \circ H) \equiv \langle \mathrm{norm}^*(F_1 \circ H), \mathrm{norm}^*(F_2 \circ H)\rangle$.

(4) $$\mathrm{norm}^*(\mathrm{Fst} \circ H) \equiv \begin{cases} H_1 & \text{if } \mathrm{norm}^*(H) \equiv \langle H_1, H_2\rangle, \\ \mathrm{Fst} & \text{if } \mathrm{norm}^*(H) \equiv \mathrm{Id}, \\ \mathrm{Fst} \circ \mathrm{norm}^*(H) & \text{otherwise.} \end{cases}$$

(5) $$\mathrm{norm}^*(\mathrm{Snd} \circ H) \equiv \begin{cases} H_2 & \text{if } \mathrm{norm}^*(H) \equiv \langle H_1, H_2\rangle, \\ \mathrm{Snd} & \text{if } \mathrm{norm}^*(H) \equiv \mathrm{Id}, \\ \mathrm{Snd} \circ \mathrm{norm}^*(H) & \text{otherwise.} \end{cases}$$

(6) $\text{norm}^*(\text{Id} \circ H) \equiv \text{norm}^*(H)$.

(7) $\text{norm}^*(c \circ H) \equiv \begin{cases} c & \text{if } \text{norm}^*(H) \equiv \text{Id}, \\ c \circ \text{norm}^*(H) & \text{otherwise}, \end{cases}$

where c is a nonspecial constant or App.

(8) $\text{norm}^*(\Lambda(F_1)) \equiv \Lambda(\text{norm}^*(F_1))$.

(9) $\text{norm}^*(\langle F_1, F_2\rangle) \equiv \langle \text{norm}^*(F_1), \text{norm}^*(F_2)\rangle$.

(10) $\text{norm}^*(c) \equiv c$, where c is a constant.

The above definition is due to transfinite induction. Let n_1, n_2, n_3 be the numbers of subterms in the form $\Lambda(H)$, $\langle H_1, H_2\rangle$, and $H_1 \circ H_2$ appearing in F, respectively. Let n_4 be na (F), where na (—) is defined as follows. For each term G, if G is in the form $(G_1 \circ G_2) \circ G_3$, then $\text{na}(G) = \text{na}(G_1 \circ G_2) + 1$, otherwise $\text{na}(G) = 0$. The above definition of $\text{norm}^*(F)$ is by transfinite induction on $\omega^3 \cdot n_1 + \omega^2 \cdot n_2 + \omega \cdot n_3 + n_4$ up to ω^4.

If we write the definition of $\text{norm}(F)$ directly, it resembles $\text{norm}^*(F)$. The difference is in the cases where $F \equiv \Lambda(F_1)$ and $F \equiv \text{App} \circ H$. When $F \equiv \Lambda(F_1)$, we have $\text{norm}(F) \equiv \Lambda(\text{norm}(F_1 \circ \langle \text{Fst}, \text{Snd}\rangle))$. When $F \equiv \text{App} \circ H$ and $\text{norm}(H)$ is not Id nor in the form $\langle H_1, H_2\rangle$, the term $\text{norm}(F)$ is not $\text{App} \circ \text{norm}(H)$ but $\text{App} \circ \langle \text{Fst} \circ \text{norm}(H), \text{Snd} \circ \text{norm}(H)\rangle$. Note that $F \xrightarrow[\text{SUB}]{*} \text{norm}^*(F)$, while $F \xrightarrow[\text{SUB}]{*} \text{norm}(F)$ is not generally true.

LEMMA 9.3. *If F is a k-stable term of* CCLβ *for some k, then*

$$\text{norm}^*(F) \equiv [\lambda\langle z_0, \cdots, z_n\rangle \cdot F^*[\langle z_0, \cdots, z_n\rangle]]^\circ$$

for every $n \geqq 0$.

Proof. By definition, $\text{norm}^*(F)$ must be in the form $c_1 \circ \cdots \circ c_m \circ H$, where $c_1, \cdots, c_m$ are constants and H is either a constant, $\Lambda(H_1)$, or $\langle H_1, H_2\rangle$. Since $F \xrightarrow[\text{SUB}]{*} \text{norm}^*(F)$, by Theorem 8.2(ii) we have $F^*[N] \equiv (\text{norm}^*(F))^*[N]$ for any term N of $\lambda\beta c$. Thus, by condition (1) of k-stability, App in $\text{norm}^*(F)$ appears only in the form $\text{App} \circ \langle H_1, H_2\rangle$. Assuming only this and condition (2) of k-stability, we will prove the lemma. The proof is by induction on the structure of $\text{norm}^*(F)$.

Case 1. H is a constant. Then $\text{norm}^*(F)$ is in the form $c_1 \circ \cdots \circ c_l \circ \pi_i^k$, where $1 \leqq i \leqq k$, $l = m - k + i$, and $c_1, \cdots, c_l$ are constants other than App and Id. This implies the lemma.

Case 2. $H \equiv \Lambda(H_1)$. Then $c_1, \cdots, c_m$ are constants other than App and Id:

$$\begin{aligned} &[\lambda\langle z_0, \cdots, z_n\rangle \cdot F^*[\langle z_0, \cdots, z_n\rangle]]^\circ \\ &\quad \equiv c_1 \circ \cdots \circ c_m \circ \Lambda([\lambda\langle z_0, \cdots, z_n, x\rangle \cdot H_1^*[\langle z_0, \cdots, z_n, x\rangle]]^\circ) \\ &\quad \equiv c_1 \circ \cdots \circ c_m \circ \Lambda(\text{norm}^*(H_1)) \quad \text{(by induction hypothesis)} \\ &\quad \equiv \text{norm}^*(F). \end{aligned}$$

Note that H_1 is $(k+1)$-stable, since F is k-stable.

Case 3. $H \equiv \langle H_1, H_2\rangle$. Then $c_1, \cdots, c_{m-1}$ are constants other than App and Id, and c_m is either a nonspecial constant or App:

$$\begin{aligned} &[\lambda\langle z_0, \cdots, z_n\rangle \cdot F^*[\langle z_0, \cdots, z_n\rangle]]^\circ \\ &\quad \equiv c_1 \circ \cdots \circ c_m \circ \langle [\lambda\langle z_0, \cdots, z_n\rangle \cdot H_1^*[\langle z_0, \cdots, z_n\rangle]]^\circ, \\ &\qquad\qquad [\lambda\langle z_0, \cdots, z_n\rangle \cdot H_2^*[\langle z_0, \cdots, z_n\rangle]]^\circ\rangle \end{aligned}$$

$\equiv c_1 \circ \cdots \circ c_m \circ \langle \text{norm}^*(H_1), \text{norm}^*(H_2) \rangle$ (by induction hypothesis)

$\equiv \text{norm}^*(F)$. □

THEOREM 9.4. *If F is an n-stable term of* CCLβ *for some n, then* $F \xrightarrow[\text{SUB}]{*} \text{norm}(F)$.

Proof. By definition, $F \xrightarrow[\text{SUB}]{*} \text{norm}^*(F)$. So, by Lemma 9.3, we have $F \xrightarrow[\text{SUB}]{*} \text{norm}(F)$. □

As explained in § 7, every term F of CCLMβ has the unique SUBM-normal form norm (F). From Theorem 9.4, similarly, it follows that every n-stable term G of CCLβ has the unique SUB-normal form norm (G).

Finally, we establish the relationship between single (—) and multi$_n$ (—).

THEOREM 9.5. (i) *For every n-ary term F of* CCLMβ, single (F) *is n-stable and* multi$_n$ (single (F)) $\equiv$ norm (F).

(ii) *For every n-stable term G of* CCLβ, single (multi$_n$ (G)) $\equiv$ norm (G).

Proof. (i) Since single $(F^*[z_1, \cdots, z_n])$ does not contain the special constants fst and snd, by Theorem 8.3 we have

$$(\text{single}(F))^*[\langle z_0, z_1, \cdots, z_n \rangle] \equiv \text{single}(F^*[z_1, \cdots, z_n]).$$

So single (F) is n-stable, and multi$_n$ (single (F)) can be defined. Moreover,

$$\begin{aligned}\text{multi}_n(\text{single}(F)) &\equiv [\lambda\langle z_1, \cdots, z_n \rangle \cdot \text{multi}(\text{single}(F^*[z_1, \cdots, z_n]))]^\circ \\ &\equiv [\lambda\langle z_1, \cdots, z_n \rangle \cdot F^*[z_1, \cdots, z_n]]^\circ \quad \text{(by Lemma 9.1(ii))} \\ &\equiv \text{norm}(F).\end{aligned}$$

(ii) Since $(\text{multi}_n(G))^*[z_1, \cdots, z_n] \equiv \text{multi}(G^*[\langle z_0, z_1, \cdots, z_n \rangle])$ by Theorem 6.3, we get

$$\begin{aligned}\text{single}(\text{multi}_n(G)) &\equiv [\lambda\langle z_0, z_1, \cdots, z_n \rangle \cdot \text{single}(\text{multi}(G^*[\langle z_0, z_1, \cdots, z_n \rangle]))]^\circ \\ &\equiv [\lambda\langle z_0, z_1, \cdots, z_n \rangle \cdot G^*[\langle z_0, z_1, \cdots, z_n \rangle]]^\circ \quad \text{(by Lemma 9.1(i))} \\ &\equiv \text{norm}(G) \quad \text{(by Lemma 9.3).}\end{aligned}$$ □

Theorem 9.5 means that the equivalence classes generated by $\xrightarrow[\text{SUBM}]{*}$ on n-ary terms of CCLMβ exactly correspond to the equivalence classes generated by $\xrightarrow[\text{SUB}]{}$ on n-stable terms of CCLβ through the maps single (—) and multi$_n$ (—). From this and theorem 9.2, we conclude that CCLMβ is embedded into CCLβ by single(—).

Acknowledgments. The authors are grateful to the referees for useful comments.

REFERENCES

[1] J. BACKUS, *Can programming be liberated from the von Neumann style? A functional style and its algebra of programs*, Comm. ACM, 21 (1978), pp. 613-641.

[2] H. P. BARENDREGT, *The Lambda Calculus—Its Syntax and Semantics*, revised ed., North-Holland, Amsterdam, 1984.

[3] G. COUSINEAU, P.-L. CURIEN, AND M. MAUNY, *The categorical abstract machine*, Sci. Comput. Programming, 8 (1987), pp. 173-202.

[4] P.-L. CURIEN, *Categorical combinators*, Inform. and Control, 69 (1986), pp. 188-254.

[5] ———, *Categorical Combinators, Sequential Algorithms and Functional Programming*, Pitman, London, 1986.

[6] ———, *The strong calculus of closures, or $\lambda\sigma$-calculus*, Preprint, 1989.

[7] Y. FUTAMURA, *Partial computation of programs*, Lecture Notes in Computer Science, 147, Springer-Verlag, Berlin, New York, 1983, pp. 1-35.

[8] T. HARDIN AND A. LAVILLE, *Proof of termination of the rewriting system* SUBST *on* CCL, Theoret. Comput. Sci., 46 (1986), pp. 305-312.

[9] T. HARDIN, *Confluence results for the pure strong categorical logic* CCL. *λ-calculi as subsystems of* CCL, Theoret. Comput. Sci., 65 (1989), pp. 291-342.

[10] S. HAYASHI, *Adjunction of semifunctors: categorical structures in nonextensional lambda calculus*, Theoret. Comput. Sci., 41 (1985), pp. 95-104.

[11] C. P. J. KOYMANS, *Models of the lambda calculus*, Inform. and Control, 52 (1982), pp. 306-332.

[12] J. LAMBEK, *Functional completeness of cartesian categories*, Ann. Math. Logic, 6 (1974), pp. 259-292.

[13] J. LAMBEK AND P. J. SCOTT, *Introduction to Higher Order Categorical Logic*, Cambridge University Press, Cambridge, 1986.

[14] F. W. LAWVERE, *Functorial semantics of algebraic theories*, Proc. Nat. Acad. Sci. U.S.A., 50 (1963), pp. 869-872.

[15] R. D. LINS, *On the efficiency of categorical combinators as a rewriting system*, Software Pract. Exper., 17 (1987), pp. 547-559.

[16] A. R. MEYER, *What is a model of the lambda calculus?*, Inform. and Control, 52 (1982), pp. 87-122.

[17] A. OBTUŁOWICZ AND A. WIWEGER, *Categorical, functorial and algebraic aspects of the type-free lambda calculus*, Universal Algebra and Applications, Banach Center Publications, Vol. 9, 1982, PWN, Warsaw, pp. 399-422.

[18] D. S. SCOTT, *Relating theories of the λ-calculus*, in To H. B. Curry: Essays on Combinatory Logic, Lambda-Calculus and Formalism, J. P. Seldin and J. Hindley, eds., Academic Press, New York, London, 1980, pp. 403-450.

[19] H. YOKOUCHI, *Retraction map categories and their applications to the construction of lambda calculus models*, Inform. and Control, 71 (1986), pp. 33-86.

[20] ———, *Church-Rosser Theorem for a rewriting system on categorical combinators*, Theoret. Comput. Sci., 65 (1989), pp. 271-290.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 98-123, February 1990

006

THE INVERSES OF BLOCK HANKEL AND BLOCK TOEPLITZ MATRICES*

GEORGE LABAHN†, DONG KOO CHOI‡, AND STAN CABAY§

Abstract. A set of new formulae for the inverse of a block Hankel (or block Toeplitz) matrix is given. The formulae are expressed in terms of certain matrix Padé forms, which approximate a matrix power series associated with the block Hankel matrix.

By using Frobenius-type identities between certain matrix Padé forms, the inversion formulae are shown to generalize the formulae of Gohberg-Heinig and, in the scalar case, the formulae of Gohberg-Semencul and Gohberg-Krupnik.

The new formulae have the significant advantage of requiring only that the block Hankel matrix itself be nonsingular. The other formulae require, in addition, that certain submatrices be nonsingular.

Since effective algorithms for computing the required matrix Padé forms are available, the formulae are practical. Indeed, some of the algorithms allow for the efficient calculation of the inverse not only of the given block Hankel matrix, but also of any nonsingular block principal minor.

Keywords. Hankel matrix, Toeplitz matrix, Padé fraction, power series, Padé form, Yule-Walker equation

AMS(MOS) subject classifications. primary, 15A09; secondary, 41A21

1. Introduction.

Let

$$H_{m,n} = \begin{bmatrix} a_{m-n+1} & \cdots & a_m \\ \vdots & & \vdots \\ a_m & \cdots & a_{m+n-1} \end{bmatrix} \tag{1.1}$$

be a nonsingular block Hankel matrix with coefficients from the ring of $p \times p$ matrices over a field.[1] The special structure of Hankel matrices has resulted in a number of closed formulae for the inverse of $H_{m,n}$.

When $p = 1$ (the scalar case) well-known formulae of Gohberg and Semencul [14] give $H_{m,n}^{-1}$ in terms of only the first and last columns of the inverse. Gohberg and Krupnik [15] give a formula for the inverse in terms of the last two columns of $H_{m,n}^{-1}$. Ben-Artzi and Shalom [3] give a series of inverse formulae, including one for determining the inverse once two adjacent columns, along with the last column, of the inverse are known.

When $p > 1$, additional problems are encountered in obtaining a closed formula for the inverse of a block Hankel matrix. When the coefficients of $H_{m,n}$ come from a noncommutative algebra there are closed formulae due to Gohberg and Heinig [16]. These are given provided the first and last columns together with the first and last rows of the inverse are known.

All of the above formulae depend on the ability to perform certain bordering operations that lend themselves well to matrices with a Hankel structure. However, these bordering operations require the imposition of certain additional restrictions on

* Received by the editors January 14, 1988; accepted for publication (in revised form) May 24, 1989.
† Department of Computing Science, University of Waterloo, Waterloo, Ontario, Canada N2L 3G1.
‡ Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1.
§ Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1. The research of this author was partially supported by Natural Sciences and Engineering Council of Canada grant A8035.

[1] All results hold, with minor modifications, for block Toeplitz matrices.

$H_{m,n}$. For the Gohberg-Krupnik formula, the matrix $H_{m-1,n-1}$ must also be nonsingular; whereas, for the Gohberg-Semencul and Gohberg-Heinig formulae, the matrix $H_{m,n-1}$ must be nonsingular. Inverse formulae are then also given for the relevant submatrices.

In the case of the scalar Gohberg-Semencul formulae, there is a standard technique for overcoming the extra requirements. When $H_{m,n}$ is nonsingular but $H_{m,n-1}$ is singular, a larger nonsingular Hankel matrix, $H_{m,n+1}$, is created. An inverse formula is then obtained by using the formulae of Gohberg-Semencul for $H_{m,n}$ and $H_{m,n+1}$ (cf. Gohberg and Semencul [14], or Iohvidov [19]). For the nonscalar case, however, there is no known similar method for overcoming the added restriction in the Gohberg-Heinig formulae.

The primary contribution of this paper is a set of new closed formulae for $H_{m,n}^{-1}$. By avoiding bordering techniques, we require only that $H_{m,n}$ be nonsingular. When $p=1$, one of the formulae agrees with that obtained by Choi [12].

The representations for $H_{m,n}^{-1}$ depend on the concept of a matrix Padé form (Labahn and Cabay [22]) for the matrix polynomial

$$A(z)=\sum_{i=0}^{m+n} a_i z^i. \tag{1.2}$$

These matrix Padé forms are determined from solutions to equations of a Yule-Walker type. Central to our approach are commutativity relationships that are shown to exist between certain matrix Padé forms. These commutativity relationships allow us to overcome the traditional limitations imposed when using bordering techniques. Indeed, the conditions that we impose are both necessary and sufficient for the existence of an inverse.

When we add the condition that $H_{m,n-1}$ also be nonsingular, certain Frobenius-type identities for matrix Padé forms are used to show that our formulae yield the formulae of Gohberg and Heinig. On the other hand, when we add the condition that $H_{m-1,n-1}$ be nonsingular, a different set of Frobenius-type identities applied to our results yields inverse formulae, which in the scalar case corresponds to the Gohberg-Krupnik formulae. Finally, using somewhat different techniques, we show how our inverse formulae provide natural generalizations of the results of Ben-Artzi and Shalom to the nonscalar case.

A major advantage of a closed inverse formula is that it allows for efficient algorithms to calculate the inverses of Hankel matrices. This efficiency comes both in the cost complexity of calculating the inverse and also in the amount of storage required for the final result.

When our inverse formulae are used in conjunction with the MPADE algorithm of Labahn and Cabay [22], we obtain an algorithm for calculating $H_{m,n}^{-1}$. This algorithm has many advantages for our situation. It is successful without any preconditions placed on the original power series. As a by-product, we obtain inverses for all the principal minors of $H_{m,n}$ that are nonsingular. Also, it is iterative on n, allowing cost savings in implementation. The complexity of the MPADE algorithm is generically $O(p^3n^2)$, although there are pathological cases where it can be as high as $O(p^3n^3)$ (for example, when all the principal minors of $H_{m,n}$ are singular). This compares with other nonscalar methods (cf. Akaike [1], Watson [31], Rissanen [27], Bose and Basu [5]) that are also of complexity $O(p^3n^2)$, but that succeed only when all principal minors are nonsingular. In the scalar case, however, the cost complexity of MPADE is $O(n^2)$, regardless of the types of singularities found in $H_{m,n}$. This compares favorably with the method described by Rissanen [28] that is of complexity $O(n^2)$ and succeeds in the degenerate case. The $O(n^2)$ methods of Trench [30], Watson [31], Zohar [33],

and Kailath, Kung, and Morf [20], on the other hand, fail whenever a principal minor of $H_{m,n}$ is singular.

When fast polynomial multiplication methods are available, in the scalar case, the required Padé forms can be calculated by the off-diagonal algorithm of Cabay and Choi [11] with a complexity of $O(n \log^2 n)$. The algorithm is also iterative on n and produces the inverses of some of the nonsingular principal minors as a bi-product. As a result of this, and some other factors, the performance is better than the $O(n \log^2 n)$ method of Brent, Gustavson, and Yun [6] and Sugiyama [29], both of which also succeed in the degenerate case. The $O(n \log^2 n)$ methods of Bitmead and Anderson [4], Ammar and Gragg [2], and de Hoog [18], on the other hand, succeed only in the nondegenerate case.

In the nonscalar case, fast algorithms can also be used to calculate the required Padé forms, but under some restrictions. If the block matrix is positive definite (or, more generally, if the associated power series is nearly-normal (cf. [21])), for example, and fast polynomial multiplication is allowed, then the inverse formulae can be calculated using the fast algorithm of Labahn [21] with complexity $O(p^3 \cdot n \log^2 n)$. This algorithm is also iterative and calculates the inverses of some of the nonsingular principal minors as a bi-product. The algorithm of Bitmead and Anderson, generalized to the nonscalar case using the formulae of Gohberg and Heinig, is also of complexity $O(p^3 \cdot n \log^2 n)$, but works only in the normal case.

For purposes of presentation, we adopt the following notation. We let D denote the noncommutative ring of $p \times p$ matrices over a field.[2] The domain of formal power series with coefficients over D and indeterminate z is denoted by $D[[z]]$. For any $A(z) \in D[[z]]$, $A(z)$ is formally represented by

$$A(z) = \sum_{i=0}^{\infty} a_i z^i, \tag{1.3}$$

where the coefficients $a_i \in D$ are always written in lower case. The domain of polynomials (finite power series) over D with indeterminant z is denoted by $D[z]$. Any polynomial $P_n(z) \in D[z]$ is represented formally by

$$P_n(z) = \sum_{i=0}^{n} p_i z^i, \tag{1.4}$$

where again the coefficients $p_i \in D$ are written in lower case. The degree of $P_n(z)$ (i.e., the largest i such that $p_i \neq 0$) is denoted by $\partial(P_n(z))$.

2. Matrix Padé forms. The inversion formulae derived in §§ 3 and 4 depend on the concept of a matrix Padé form for a matrix power series. This is a multidimensional generalization of scalar Padé forms (cf. Gragg [17]). Let

$$A(z) = \sum_{i=0}^{\infty} a_i z^i \in D[[z]] \tag{2.1}$$

be a formal power series with coefficients from the ring D of $p \times p$ matrices over some field. For nonnegative integers m and n, let

$$U_m(z) = \sum_{i=0}^{m} u_i z^i, \qquad V_n(z) = \sum_{i=0}^{n} v_i z^i \quad \in \quad D[z] \tag{2.2}$$

be $p \times p$ matrix polynomials.

[2] All the results of this paper can be presented in the more general setting where D is an arbitrary noncommutative algebra.

DEFINITION 2.1 (Labahn and Cabay [22]). The triple $(U_m(z), V_n(z), W(z))$ is defined to be a **Right Matrix Padé Form** (RMPFo) of type (m, n) for the power series $A(z)$ if

(I) $\partial(U_m(z)) \leqq m$, $\partial(V_n(z)) \leqq n$,

(II) $A(z) \cdot V_n(z) - U_m(z) = z^{m+n+1} W(z)$, where $W(z) \in D[[z]]$, and

(III) The columns of $V_n(z)$ are linearly independent over the field.[3]

The matrices $U_m(z)$, $V_n(z)$, and $W(z)$ are called the **right numerator, denominator**, and **residual** (all of type (m, n)), respectively. □

There is an equivalent definition for a left matrix Padé form (LMPFo). In condition (II), multiplication on the right by $V_n(z)$ is replaced by multiplication on the left. In addition, condition (III) is replaced by

(III) the rows of $V_n(z)$ are linearly independent over the field.

Condition (II) can be written as follows:

$$\begin{bmatrix} a_{-n} & \cdots & a_0 \\ \vdots & & \vdots \\ a_{m-n} & \cdots & a_m \end{bmatrix} \cdot \begin{bmatrix} v_n \\ \vdots \\ v_0 \end{bmatrix} = \begin{bmatrix} u_0 \\ \vdots \\ u_m \end{bmatrix} \tag{2.3}$$

and

$$\begin{bmatrix} a_{m-n+1} & \cdots & a_m & a_{m+1} \\ \vdots & & & \vdots \\ a_m & a_{m+1} & \cdots & a_{m+n} \end{bmatrix} \cdot \begin{bmatrix} v_n \\ \vdots \\ v_0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{2.4}$$

Here $a_i = 0$ for $i < 0$. The matrix polynomial $V_n(z)$ can be determined by solving (2.4), and then $U_m(z)$ can be obtained from (2.3).

THEOREM 2.2 (Existence of Matrix Padé Forms). *For any matrix power series $A(z)$ and for any pair of nonnegative integers (m, n), there exists an* RMPFo *and an* LMPFo *of type (m, n).*

Proof. The result follows from (2.3) and (2.4) by comparing the number of equations with the number of unknowns. For details see [22]. □

To distinguish between matrix Padé forms of different types, we introduce the following notation. For a given pair of positive integers (m, n), the triples $(U_m(z), V_n(z), W(z))$ and $(U_m^*(z), V_n^*(z), W^*(z))$ denote an RMPFo and an LMPFo, respectively, of type (m, n) for $A(z)$. For the same (m, n), an RMPFo and an LMPFo of type $(m-1, n-1)$ for $A(z)$ are represented, respectively, by $(P_{m-1}(z), Q_{n-1}(z), R(z))$ and $(P_{m-1}^*(z), Q_{n-1}^*(z), R^*(z))$. For these Padé forms, collectively, condition (II) becomes

$$A(z) V_n(z) - U_m(z) = z^{m+n+1} \cdot W(z), \tag{2.5}$$

$$V_n^*(z) A(z) - U_m^*(z) = z^{m+n+1} \cdot W^*(z), \tag{2.6}$$

$$A(z) Q_{n-1}(z) - P_{m-1}(z) = z^{m+n-1} \cdot R(z), \tag{2.7}$$

$$Q_{n-1}^*(z) A(z) - P_{m-1}^*(z) = z^{m+n-1} \cdot R^*(z). \tag{2.8}$$

In § 3, in the case that $H_{m,n}$ is nonsingular, the inverse is given in terms of these four matrix Padé forms.

THEOREM 2.3. *For a pair of positive integers (m, n), the following statements are equivalent:*

$$\det(H_{m,n}) \neq 0, \tag{2.9}$$

[3] When the leading term v_0 is nonsingular, then in [22] a RMPFo is called a Right Matrix Padé Fraction (RMPFr).

$$\det(r_0) \neq 0 \quad \text{and} \quad \det(v_0) \neq 0, \tag{2.10}$$

$$\det(r_0^*) \neq 0 \quad \text{and} \quad \det(v_0^*) \neq 0. \tag{2.11}$$

Proof. That (2.9) implies (2.10) and (2.9) implies (2.11) was proved in [22], and so we show only the converse here. To see that (2.10) implies (2.9), let $X = (x_1, \cdots, x_n)$ be a vector of length np and suppose that

$$X \cdot H_{m,n} = 0. \tag{2.12}$$

We shall show that $X = 0$. We accomplish this by showing that (2.12) implies that $x_n = 0$ and

$$(0, x_1, \cdots, x_{n-1}) \cdot H_{m,n} = 0. \tag{2.13}$$

By repeated application of this property, it then follows that $x_{n-1} = \cdots = x_1 = 0$, and so $X = 0$.

First observe that equating coefficients of z^i, for $m+1 \leqq i \leqq m+n$, in (2.5) yields

$$H_{m,n} \cdot \begin{bmatrix} v_n \\ \vdots \\ v_1 \end{bmatrix} = -\begin{bmatrix} a_{m+1} \\ \vdots \\ a_{m+n} \end{bmatrix} v_0 \tag{2.14}$$

where v_0 is invertible since we are assuming statement (2.10). Similarly, equating coefficients of z^i, for $m \leqq i \leqq m+n-1$, in (2.7) yields

$$H_{m,n} \cdot \begin{bmatrix} q_{n-1} \\ \vdots \\ q_0 \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ r_0 \end{bmatrix} \tag{2.15}$$

where, by assumption, r_0 is invertible. From (2.12) and (2.15), it follows that

$$x_n \cdot r_0 = X \cdot \begin{bmatrix} 0 \\ \vdots \\ 0 \\ r_0 \end{bmatrix} = X \cdot H_{m,n} \begin{bmatrix} q_{n-1} \\ \vdots \\ q_0 \end{bmatrix} = 0. \tag{2.16}$$

Since r_0 is invertible, it then follows that $x_n = 0$.

Having shown that $x_n = 0$, (2.12) then yields

$$(x_1, \cdots, x_{n-1}) \cdot \begin{bmatrix} a_{m-n+2} & \cdots & a_m \\ \vdots & & \vdots \\ a_m & \cdots & a_{m+n-2} \end{bmatrix} = 0. \tag{2.17}$$

But, from (2.12) and (2.14), we have

$$(x_1, \cdots, x_{n-1}, 0) \cdot \begin{bmatrix} a_{m+1} \\ \vdots \\ a_{m+n} \end{bmatrix} \cdot v_0 = -X \cdot H_{m,n} \begin{bmatrix} v_n \\ \vdots \\ v_1 \end{bmatrix} = 0. \tag{2.18}$$

Since v_0 is invertible, (2.18) implies that

$$(x_1, \cdots, x_{n-1}) \cdot \begin{bmatrix} a_{m+1} \\ \vdots \\ a_{m+n-1} \end{bmatrix} = 0. \tag{2.19}$$

Equations (2.17) and (2.19) imply that

$$(x_1, \cdots, x_{n-1}) \cdot \begin{bmatrix} a_{m-n+2} & \cdots & a_{m+1} \\ \vdots & & \vdots \\ a_m & \cdots & a_{m+n-1} \end{bmatrix} = 0, \tag{2.20}$$

which is equivalent to (2.13).

Thus, we have shown that (2.10) implies (2.9). A similar argument shows that (2.11) implies (2.9). □

Theorem 2.3 has important computational significance since the singularity of $H_{m,n}$ can be detected simply by recognizing a singular r_0 or a singular v_0. If both r_0 and v_0 are nonsingular, then we have Theorem 2.4.

THEOREM 2.4. *If* $\det(H_{m,n}) \neq 0$, *then the matrix Padé forms identified by* (2.5)-(2.8) *are unique, except for the specification of the nonsingular matrices* v_0, v_0^*, r_0, *and* r_0^*.

Proof. We refer the reader to Theorems 3.2 and 3.3 in [22] for a detailed proof of this result. □

As a consequence of Theorem 2.4, it can be assumed without loss of generality that

$$v_0 = v_0^* = r_0 = r_0^* = I. \tag{2.21}$$

This nonrestrictive assumption simplifies the presentation of subsequent results.

The key relationship between matrix Padé forms that enables the presentation of the inverse of $H_{m,n}$ in §§ 3 and 4, is given by Lemma 2.5.

LEMMA 2.5. *Let* $\det(H_{m,n}) \neq 0$. *Then the matrix Padé forms identified by* (2.5)-(2.8) *and normalized according to* (2.21) *satisfy*

$$\begin{bmatrix} Q_{n-1}^*(z) & -P_{m-1}^*(z) \\ -V_n^*(z) & U_m^*(z) \end{bmatrix} \cdot \begin{bmatrix} U_m(z) & P_{m-1}(z) \\ V_n(z) & Q_{n-1}(z) \end{bmatrix} = z^{m+n-1} \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}, \tag{2.22}$$

$$\begin{bmatrix} U_m(z) & P_{m-1}(z) \\ V_n(z) & Q_{n-1}(z) \end{bmatrix} \cdot \begin{bmatrix} Q_{n-1}^*(z) & -P_{m-1}^*(z) \\ -V_n^*(z) & U_m^*(z) \end{bmatrix} = z^{m+n-1} \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}, \tag{2.23}$$

$$\begin{bmatrix} R^*(z) & -Q_{n-1}^*(z) \\ -z^2 W^*(z) & V_n^*(z) \end{bmatrix} \cdot \begin{bmatrix} V_n(z) & Q_{n-1}(z) \\ z^2 W(z) & R(z) \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}, \tag{2.24}$$

$$\begin{bmatrix} V_n(z) & Q_{n-1}(z) \\ z^2 W(z) & R(z) \end{bmatrix} \cdot \begin{bmatrix} R^*(z) & -Q_{n-1}^*(z) \\ -z^2 W^*(z) & V_n^*(z) \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}. \tag{2.25}$$

Proof. Multiplying (2.5) on the left by $Q_{n-1}^*(z)$ and (2.8) on the right by $V_n(z)$, and subtracting the first from the second, we obtain

$$\begin{aligned} Q_{n-1}^*(z) \cdot U_m(z) - P_{m-1}^*(z) \cdot V_n(z) &= z^{m+n-1} \cdot (R^*(z) V_n(z) - z^2 Q_{n-1}^*(z) W(z)) \\ &= z^{m+n-1} r_0^* v_0 \\ &= z^{m+n-1} I. \end{aligned} \tag{2.26}$$

In (2.26), we have used the normalizing condition (2.21) and the fact that the left-hand side, and consequently the right-hand side, is a matrix polynomial of degree at most $m+n-1$.

Multiplying (2.5) on the left by $V_n^*(z)$ and (2.6) on the right by $V_n(z)$, and subtracting the second from the first, we obtain

$$\begin{aligned} -V_{n-1}^*(z) \cdot U_m(z) + U_m^*(z) \cdot V_n(z) &= z^{m+n+1} \cdot (V_n^*(z) W(z) - W^*(z) V_n(z)) \\ &= 0. \end{aligned} \tag{2.27}$$

In (2.27), the last equality is true because the left-hand side, and consequently the right-hand side, is a matrix polynomial of degree at most $m+n$.

In a similar fashion, (2.7), (2.8), and (2.21) yield

$$Q^*_{n-1}(z)P_{m-1}(z) - P^*_{m-1}(z)Q_{n-1}(z) = z^{m+n-1}\cdot(Q^*_{n-1}(z)R(z) - R^*(z)Q_{n-1}(z)) = 0; \tag{2.28}$$

whereas, (2.6), (2.7), and (2.21) give

$$-V^*_n(z)P_{m-1}(z) + U^*_m(z)Q_{n-1}(z) = z^{m+n-1}\cdot(V^*_n(z)R(z) - z^2W^*(z)Q_{n-1}) = z^{m+n-1}I. \tag{2.29}$$

Equations (2.26)-(2.29) together comprise (2.22). Equation (2.23) follows directly from (2.22), since matrix inverses are two sided.

Equations (2.26) also gives

$$z^{m+n-1}\cdot(R^*(z)V_n(z) - z^2Q^*_{n-1}(z)W(z)) = z^{m+n-1}\cdot I, \tag{2.30}$$

from which we obtain

$$R^*(z)V_n(z) - z^2Q^*_{n-1}(z)W(z) = I. \tag{2.31}$$

Similarly, from (2.27), we obtain

$$V^*_n(z)W(z) - W^*(z)V_n(z) = 0. \tag{2.32}$$

From (2.28), we obtain

$$Q^*_{n-1}(z)R(z) - R^*(z)Q_{n-1}(z) = 0, \tag{2.33}$$

and (2.29) gives

$$V^*_n(z)R(z) - z^2W^*(z)Q_{n-1} = I. \tag{2.34}$$

Equations (2.31)-(2.34) comprise (2.24). As before, (2.25) follows from (2.24), since matrix inverses are two-sided. □

3. The off-diagonal inverse formula. The main result of this paper is Theorem 3.1.

THEOREM 3.1. *Let $H_{m,n}$ be the block Hankel matrix* (1.1). *If there are* RMPFo*s and* LMPFo*s of type $(m-1, n-1)$ and (m, n) for $A(z)$ satisfying the normalizing condition* (2.21), *then $H_{m,n}$ is nonsingular with inverse*

$$H^{-1}_{m,n} = \begin{bmatrix} v_{n-1} & \cdots & v_0 \\ \vdots & \iddots & \\ v_0 & & \end{bmatrix}\begin{bmatrix} q^*_{n-1} & \cdots & q^*_0 \\ & \ddots & \vdots \\ & & q^*_{n-1} \end{bmatrix} - \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix}\begin{bmatrix} v^*_n & \cdots & v^*_1 \\ & \ddots & \vdots \\ & & v^*_n \end{bmatrix}, \tag{3.1}$$

or, equivalently,

$$H^{-1}_{m,n} = \begin{bmatrix} q_{n-1} & & \\ \vdots & \ddots & \\ q_0 & \cdots & q_{n-1} \end{bmatrix}\begin{bmatrix} v^*_{n-1} & \cdots & v^*_0 \\ \vdots & \iddots & \\ v^*_0 & & \end{bmatrix} - \begin{bmatrix} v_n & & \\ \vdots & \ddots & \\ v_1 & \cdots & v_n \end{bmatrix}\begin{bmatrix} q^*_{n-2} & \cdots & q^*_0 & 0 \\ \vdots & \iddots & & \\ q^*_0 & & & \\ 0 & & & \end{bmatrix}. \tag{3.2}$$

Proof. Using

$$U_m(z)Q^*_{n-1}(z) - P_{m-1}(z)V^*_n(z) = z^{m+n-1}I,$$

which is from (2.23), we can equate coefficients of z^i, $m \leqq i \leqq m+n-1$, to obtain

$$\begin{bmatrix} u_m & \cdots & u_{m-n+1} \\ & \ddots & \vdots \\ & & u_m \end{bmatrix}\cdot\begin{bmatrix} q^*_{n-1} & \cdots & q^*_0 \\ & \ddots & \vdots \\ & & q^*_{n-1} \end{bmatrix} - \begin{bmatrix} p_{m-1} & \cdots & p_{m-n} \\ & \ddots & \vdots \\ & & p_{m-1} \end{bmatrix}\cdot\begin{bmatrix} v^*_n & \cdots & v^*_1 \\ & \ddots & \vdots \\ & & v^*_n \end{bmatrix} = I. \tag{3.3}$$

Similarly,

$$V_n(z)Q_{n-1}^*(z) - Q_{n-1}(z)V_n^*(z) = 0,$$

also from (2.23), yields

$$\begin{bmatrix} & & v_n \\ & \iddots & \vdots \\ v_n & \cdots & v_1 \end{bmatrix} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} - \begin{bmatrix} & & q_{n-1} \\ & \iddots & \vdots \\ q_{n-1} & \cdots & q_0 \end{bmatrix} \begin{bmatrix} v_n^* & \cdots & v_1^* \\ & \ddots & \vdots \\ & & v_n^* \end{bmatrix} = 0. \tag{3.4}$$

Now, from (2.7), we obtain

$$H_{m,n} \cdot \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & \iddots & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} = \begin{bmatrix} p_{m-1} & \cdots & p_{m-n} \\ & \ddots & \vdots \\ & & p_{m-1} \end{bmatrix} - H_{m-n,n} \cdot \begin{bmatrix} & & q_{n-1} \\ & \iddots & \vdots \\ q_{n-1} & \cdots & q_0 \end{bmatrix}. \tag{3.5}$$

Observe that, for $1 \leqq i, j \leqq n$, the (i, j) component in (3.5) is obtained by equating coefficients of $z^{m+i-j-1}$ in (2.7). Similarly, (2.5) yields

$$H_{m,n} \cdot \begin{bmatrix} v_{n-1} & \cdots & v_0 \\ \vdots & \iddots & \\ v_0 & & \end{bmatrix} = \begin{bmatrix} u_m & \cdots & u_{m-n+1} \\ & \ddots & \vdots \\ & & u_m \end{bmatrix} - H_{m-n,n} \cdot \begin{bmatrix} & & v_n \\ & \iddots & \vdots \\ v_n & \cdots & v_1 \end{bmatrix}. \tag{3.6}$$

Combining (3.5) and (3.6) and using (3.3) and (3.4), it then follows that

$$\begin{aligned} H_{m,n} & \left\{ \begin{bmatrix} v_{n-1} & \cdots & v_0 \\ \vdots & \iddots & \\ v_0 & & \end{bmatrix} \cdot \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} - \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & \iddots & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} \begin{bmatrix} v_n^* & \cdots & v_1^* \\ & \ddots & \vdots \\ & & v_n^* \end{bmatrix} \right\} \\ & = \left\{ \begin{bmatrix} u_m & \cdots & u_{m-n+1} \\ & \ddots & \vdots \\ & & u_m \end{bmatrix} - H_{m-n,n} \cdot \begin{bmatrix} & & v_n \\ & \iddots & \vdots \\ v_n & \cdots & v_1 \end{bmatrix} \right\} \cdot \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\ & \quad - \left\{ \begin{bmatrix} p_{m-1} & \cdots & p_{m-n} \\ & \ddots & \vdots \\ & & p_{m-1} \end{bmatrix} - H_{m-n,n} \begin{bmatrix} & & q_{n-1} \\ & \iddots & \vdots \\ q_{n-1} & \cdots & q_0 \end{bmatrix} \right\} \cdot \begin{bmatrix} v_n^* & \cdots & v_1^* \\ & \ddots & \vdots \\ & & v_n^* \end{bmatrix} \\ & = \left\{ \begin{bmatrix} u_m & \cdots & u_{m-n+1} \\ & \ddots & \vdots \\ & & u_m \end{bmatrix} \cdot \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} - \begin{bmatrix} p_{m-1} & \cdots & p_{m-n} \\ & \ddots & \vdots \\ & & p_{m-1} \end{bmatrix} \cdot \begin{bmatrix} v_n^* & \cdots & v_1^* \\ & \ddots & \vdots \\ & & v_n^* \end{bmatrix} \right\} \\ & \quad - H_{m-n,n} \left\{ \begin{bmatrix} & & v_n \\ & \iddots & \vdots \\ v_n & \cdots & v_1 \end{bmatrix} \cdot \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \right. \\ & \quad \left. - \begin{bmatrix} & & q_{n-1} \\ & \iddots & \vdots \\ q_{n-1} & \cdots & q_0 \end{bmatrix} \cdot \begin{bmatrix} v_n^* & \cdots & v_1^* \\ & \ddots & \vdots \\ & & v_n^* \end{bmatrix} \right\} \\ & = I. \end{aligned} \tag{3.7}$$

Thus, $H_{m,n}$ is nonsingular with the inverse given by (3.1).

The second formula (3.2) for the inverse is proved using (2.6), (2.8), and the second column of (2.23). □

Remark 1. In the scalar case, (3.1) was first obtained by Choi [12].

Remark 2. The assumptions of Theorem 3.1 can be equivalently replaced by the requirement that we obtain solutions to

$$H_{m,n} \cdot [q_{n-1}, \cdots, q_0]^t = [0, \cdots, 0, I]^t, \tag{3.8}$$

$$[q_{n-1}^*, \cdots, q_0^*] \cdot H_{m,n} = [0, \cdots, 0, I], \tag{3.9}$$

$$H_{m,n} \cdot [v_n, \cdots, v_1]^t = -[a_{m+1}, \cdots, a_{m+n-1}, a_{m+n}]^t, \tag{3.10}$$

$$[v_n^*, \cdots, v_1^*] \cdot H_{m,n} = -[a_{m+1}, \cdots, a_{m+n-1}, a_{m+n}] \tag{3.11}$$

where a_{m+n} can be any $p \times p$ matrix. Equations (3.10) and (3.11) are block versions of the Yule-Walker equations.

4. The antidiagonal inverse formula. Theorem 3.1 provides inverse formulae for the block Hankel matrix $H_{m,n}$ in terms of RMPFo and LMPFo of type $(m-1, n-1)$ and (m, n) for the associated matrix polynomial $A(z)$. There are some algorithms (cf. [6], [24], [29]) that calculate Padé forms along an antidiagonal, rather than along an off-diagonal path of the Padé table. For this reason, it is useful to provide inverse formulae in terms of RMPFos and LMPFos of type $(m-1, n)$ and $(m, n-1)$ for $A(z)$.

Let $(E_m(z), F_{n-1}(z), G(z))$ and $(E_m^*(z), F_{n-1}^*(z), G^*(z))$ be an RMPFo and an LMPFo, respectively, of type $(m, n-1)$ for $A(z)$. Also, let $(B_{m-1}(z), C_n(z), D(z))$ and $(B_{m-1}^*(z), C_n^*(z), D^*(z))$ be an RMPFo and an LMPFo, respectively, of type $(m-1, n)$ for $A(z)$. Then, the following equations are satisfied:

$$A(z)F_{n-1}(z) - E_m(z) = z^{m+n}G(z), \tag{4.1}$$

$$F_{n-1}^*(z)A(z) - E_m^*(z) = z^{m+n}G^*(z), \tag{4.2}$$

$$A(z)C_n(z) - B_{m-1}(z) = z^{m+n}D(z), \tag{4.3}$$

$$C_n^*(z)A(z) - B_{m-1}^*(z) = z^{m+n}D^*(z). \tag{4.4}$$

COROLLARY 4.1. *Let $H_{m,n}$ be the block Hankel matrix* (1.1). *Then the following are equivalent*:

$$\det(H_{m,n}) \neq 0, \tag{4.5}$$

$$\det(e_m) \neq 0 \quad \textit{and} \quad \det(c_n) \neq 0, \tag{4.6}$$

$$\det(e_m^*) \neq 0 \quad \textit{and} \quad \det(c_n^*) \neq 0. \tag{4.7}$$

If any (and therefore all) of (4.5), (4.6), or (4.7) hold, then the inverse is given by

$$H_{m,n}^{-1} = \begin{bmatrix} c_n & & \\ \vdots & \ddots & \\ c_1 & \cdots & c_n \end{bmatrix} \begin{bmatrix} f_{n-1}^* & \cdots & f_0^* \\ \vdots & \iddots & \\ f_0^* & & \end{bmatrix} - \begin{bmatrix} 0 & & & \\ f_{n-1} & \ddots & & \\ \vdots & \ddots & \ddots & \\ f_1 & \cdots & f_{n-1} & 0 \end{bmatrix} \begin{bmatrix} c_{n-1}^* & \cdots & c_0^* \\ \vdots & \iddots & \\ c_0^* & & \end{bmatrix}, \tag{4.8}$$

or, equivalently,

$$H_{m,n}^{-1} = \begin{bmatrix} f_{n-1} & \cdots & f_0 \\ \vdots & \iddots & \\ f_0 & & \end{bmatrix} \begin{bmatrix} c_n^* & \cdots & c_1^* \\ & \ddots & \vdots \\ & & c_n^* \end{bmatrix} - \begin{bmatrix} c_{n-1} & \cdots & c_0 \\ \vdots & \iddots & \\ c_0 & & \end{bmatrix} \begin{bmatrix} 0 & f_{n-1}^* & \cdots & f_1^* \\ & \ddots & \ddots & \vdots \\ & & \ddots & f_{n-1}^* \\ & & & 0 \end{bmatrix} \tag{4.9}$$

where we have normalized the Padé forms so that

$$e_m = e_m^* = c_n = c_n^* = I. \tag{4.10}$$

Proof. Let $a_i^{\#} = a_{2m-i}$, for $0 \leqq i \leqq m+n$, and define a truncated power series $A^{\#}(z) = \sum_{i=0}^{m+n} a_i^{\#} z^i$. Observe that, if

$$H_{m,n}^{\#} = \begin{bmatrix} a_{m-n+1}^{\#} & \cdots & a_m^{\#} \\ \vdots & & \vdots \\ a_m^{\#} & \cdots & a_{m+n-1}^{\#} \end{bmatrix}, \tag{4.11}$$

then

$$H_{m,n}^{\#} = J \cdot H_{m,n} \cdot J \tag{4.12}$$

where

$$J = \begin{bmatrix} 0 & I \\ I & 0 \end{bmatrix}.$$

Equating coefficients of z^i, for $m \leqq i \leqq m+n-1$, in (4.1), we obtain

$$H_{m,n} \begin{bmatrix} f_{n-1} \\ \vdots \\ f_0 \end{bmatrix} = \begin{bmatrix} e_m \\ 0 \\ \vdots \\ 0 \end{bmatrix}. \tag{4.13}$$

From (4.12) and (4.13), it then follows that

$$H_{m,n}^{\#} \begin{bmatrix} f_0 \\ \vdots \\ f_{n-1} \end{bmatrix} = \begin{bmatrix} 0 \\ \vdots \\ 0 \\ e_m \end{bmatrix}. \tag{4.14}$$

Thus,

$$Q_{n-1}(z) = \sum_{i=0}^{n-1} f_{n-1+i} z^i \tag{4.15}$$

is a right denominator of type $(m-1, n-1)$ for $A^{\#}(z)$. Similarly, (4.2) yields

$$[f_0^*, \cdots, f_{n-1}^*] H_{m,n}^{\#} = [0, \cdots, 0, e_m^*], \tag{4.16}$$

and so

$$Q_{n-1}^*(z) = \sum_{i=0}^{n-1} f_{n-1+i}^* z^i \tag{4.17}$$

is a left denominator of type $(m-1, n-1)$ for $A^{\#}(z)$.

Next, from (4.3), we obtain

$$H_{m,n} \begin{bmatrix} c_{n-1} \\ \vdots \\ c_0 \end{bmatrix} = \begin{bmatrix} a_{m-n} \\ \vdots \\ a_{m-1} \end{bmatrix} c_n, \tag{4.18}$$

and so (4.12) then gives

$$H_{m,n}^{\#} \begin{bmatrix} c_0 \\ \vdots \\ c_{n-1} \end{bmatrix} = \begin{bmatrix} a_{m+1}^{\#} \\ \vdots \\ a_{m+n}^{\#} \end{bmatrix} c_n. \tag{4.19}$$

Thus,

$$V_n(z) = \sum_{i=0}^{n} c_{n-i} z^i \tag{4.20}$$

is a right denominator of type (m, n) for $A^{\#}(z)$. Similarly, (4.4) can be used to obtain

$$[c_0^*, \cdots, c_{n-1}^*] H_{m,n}^{\#} = c_n^* [a_{m+1}^{\#}, \cdots, a_{m+n}^{\#}], \tag{4.21}$$

and so

$$V_n^*(z) = \sum_{i=0}^{n} c_{n-i}^* z^i \tag{4.22}$$

is a left denominator of type (m, n) for $A^{\#}(z)$. Since $\det(H_{m,n}^{\#}) \neq 0$ if and only if $\det(H_{m,n}) \neq 0$, the equivalence of (4.5)-(4.7) now follows from the equivalence of (2.9)-(2.11).

To prove (4.8), normalize according to (4.10) and substitute (4.15), (4.17), (4.20), and (4.22) into (3.1) to obtain

$$H_{m,n}^{\#-1} = \begin{bmatrix} c_1 & \cdots & c_n \\ \vdots & \iddots & \\ c_n & & \end{bmatrix} \begin{bmatrix} f_0^* & \cdots & f_{n-1}^* \\ & \ddots & \vdots \\ & & f_0^* \end{bmatrix} - \begin{bmatrix} f_1 & \cdots & f_{n-1} & 0 \\ \vdots & \iddots & & \\ f_{n-1} & \iddots & & \\ 0 & & & \end{bmatrix} \begin{bmatrix} c_0^* & \cdots & c_{n-1}^* \\ & \ddots & \vdots \\ & & c_0^* \end{bmatrix}. \tag{4.23}$$

By using (4.12), (4.8) follows immediately from (4.23). In a similar fashion, (4.9) can be obtained using (3.2). □

5. The Gohberg–Heinig inverse formulae. In this and the next section, we compare our inverse formulae (3.1) and (3.2) with other similar well-known formulae. In terms of matrix Padé forms of type $(m-1, n-1)$ and $(m, n-1)$, the inverse of $H_{m,n}$ is given by Corollary 5.1.

COROLLARY 5.1. *Let the matrix Padé forms identified by* (2.7), (2.8), (4.1), *and* (4.2) *be given. Then the following statements are equivalent*:

$$\det(H_{m,n-1}) \neq 0 \quad \textit{and} \quad \det(H_{m,n}) \neq 0, \tag{5.1}$$

$$\det(r_0) \neq 0 \quad \textit{and} \quad \det(f_0) \neq 0, \tag{5.2}$$

$$\det(r_0^*) \neq 0 \quad \textit{and} \quad \det(f_0^*) \neq 0. \tag{5.3}$$

In addition, if any (*and therefore all*) *of conditions* (5.1), (5.2), *or* (5.3) *are satisfied, then*

$$\begin{aligned} H_{m,n}^{-1} = & \begin{bmatrix} f_{n-1} & \cdots & f_0 \\ \vdots & \iddots & \\ f_0 & & \end{bmatrix} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\ & - \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} \begin{bmatrix} 0 & f_{n-1}^* & \cdots & f_1^* \\ & \ddots & \ddots & \vdots \\ & & \ddots & f_{n-1}^* \\ & & & 0 \end{bmatrix} \end{aligned} \tag{5.4}$$

where the Padé forms have been normalized by

$$r_0 = r_0^* = f_0 = f_0^* = I. \tag{5.5}$$

Proof. We first show that (5.1) implies (5.2). Since $\det(H_{m,n}) \neq 0$, Theorem 2.3 implies that $\det(r_0) \neq 0$. Since $\det(H_{m,n-1}) \neq 0$, Theorem 2.3 also implies that $\det(f_0) \neq 0$. Therefore (5.1) implies (5.2). In a similar fashion, (5.1) implies (5.3).

To show that (5.2) implies (5.1), let

$$U_m(z) = E_m(z) - z \cdot P_{m-1}(z) r_0^{-1} g_0, \tag{5.6}$$

$$V_n(z) = F_{n-1}(z) - z \cdot Q_{n-1}(z) r_0^{-1} g_0. \tag{5.7}$$

Then, $\partial(U_m(z)) \leqq m$ and $\partial(V_n(z)) \leqq n$. Also,

$$\begin{aligned} A(z)V_n(z) - U_m(z) &= \{A(z)F_{n-1}(z) - E_m(z)\} - z\{A(z)Q_{n-1}(z) - P_{m-1}(z)\} r_0^{-1} g_0 \\ &= z^{m+n}\{G(z) - R(z) r_0^{-1} g_0\} \qquad (5.8) \\ &= z^{m+n+1} W(z) \end{aligned}$$

where

$$W(z) = z^{-1}\{G(z) - R(z) r_0^{-1} g_0\} \in D[[z]]. \tag{5.9}$$

Finally, the columns of $V_n(z)$ are linearly independent since from (5.7) $v_0 = f_0$, and by assumption f_0 is nonsingular. Thus, $(U_m(z), V_n(z), W(z))$ is an RMPFo of type (m, n) for $A(z)$, satisfying $\det(v_0) \neq 0$. From Theorem 2.3, it follows that $H_{m,n}$ is nonsingular since both $\det(r_0) \neq 0$ and $\det(v_0) \neq 0$. To see that $H_{m,n-1}$ is also nonsingular, observe that

$$\begin{bmatrix} a_{m-n+1} & \cdots & a_m \\ \vdots & & \vdots \\ a_m & \cdots & a_{m+n-1} \end{bmatrix} \begin{bmatrix} I & & f_{n-1} \\ & \ddots & \vdots \\ & I & f_1 \\ 0 & \cdots 0 & f_0 \end{bmatrix} = \begin{bmatrix} a_{m-n+1} & \cdots & a_{m-1} & e_m \\ a_{m-n+2} & \cdots & a_m & 0 \\ \vdots & & \vdots & \vdots \\ a_m & \cdots & a_{m+n-2} & 0 \end{bmatrix} \tag{5.10}$$

where the last column is determined by equating coefficients of z^i, for $m \leqq i \leqq m+n-1$ in (4.1). Thus, $\det(H_{m,n}) \neq 0$ and $\det(f_0) \neq 0$ implies that $\det(H_{m,n-1}) \neq 0$. Thus, (5.2) implies (5.1).

In a similar fashion, by defining

$$U_m^*(z) = E_m^*(z) - z g_0^* r_0^{*-1} P_{m-1}^*(z), \tag{5.11}$$

$$V_n^*(z) = F_{n-1}^*(z) - z g_0^* r_0^{*-1} Q_{n-1}^*(z), \tag{5.12}$$

it can be shown that (5.3) implies (5.1).

To obtain the inverse formula, substitution of (5.6), (5.7), (5.11), and (5.12), after normalization by (5.5), into equation (3.1) gives

$$\begin{aligned} H_{m,n}^{-1} = &\begin{bmatrix} f_{n-1} & \cdots & f_0 \\ \vdots & \iddots & \\ f_0 & & \end{bmatrix} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} - \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} \begin{bmatrix} 0 & f_{n-1}^* & \cdots & f_1^* \\ & & \ddots & \vdots \\ & & \ddots & f_{n-1}^* \\ & & & 0 \end{bmatrix} \\ &+ \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} (g_0 - g_0^*) \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix}. \end{aligned} \tag{5.13}$$

But, (4.1) and (4.2) imply that

$$E_m^*(z)F_{n-1}(z) - F_{n-1}^*(z)E_m(z) = z^{m+n}\{F_{n-1}^*(z)G(z) - G^*(z)F_{n-1}(z)\}. \tag{5.14}$$

Consequently,

$$F^*_{n-1}(z)G(z) - G^*(z)F_{n-1}(z) = 0 \tag{5.15}$$

and, in particular,

$$g_0 = g^*_0. \tag{5.16}$$

Thus, (5.13) is exactly (5.4), since the last product cancels. □

Remark 1. Corollary 5.1 can be proved directly from (2.7), (2.8), (4.1), and (4.2). Indeed, using the same arguments as in Lemma 2.5, we can obtain

$$\begin{bmatrix} r_0^{*-1}Q^*_{n-1}(z) & -r_0^{*-1}P^*_{m-1}(z) \\ -f_0^{*-1}F^*_{n-1}(z) & f_0^{*-1}E^*_m(z) \end{bmatrix} \cdot \begin{bmatrix} E_m(z)f_0^{-1} & P_{m-1}(z)r_0^{-1} \\ F_{n-1}(z)f_0^{-1} & Q_{n-1}(z)r_0^{-1} \end{bmatrix} = z^{m+n-1}\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \tag{5.17}$$

and the commutative relationship

$$\begin{bmatrix} E_m(z)f_0^{-1} & P_{m-1}(z)r_0^{-1} \\ F_{n-1}(z)f_0^{-1} & Q_{n-1}(z)r_0^{-1} \end{bmatrix} \cdot \begin{bmatrix} r_0^{*-1}Q^*_{n-1}(z) & -r_0^{*-1}P^*_{m-1}(z) \\ -f_0^{*-1}F^*_{n-1}(z) & f_0^{*-1}E^*_m(z) \end{bmatrix} = z^{m+n-1}\begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix}. \tag{5.18}$$

Consequently, we can normalize our Padé forms according to (5.5) and the formulae will follow in a fashion similar to the proof of Theorem 3.1.

The actual proof, in addition to being simpler, serves to illustrate the existence of Frobenius-type relationships (generalized from the scalar case (cf. Gragg [17]) to the matrix case) between matrix Padé forms of types (m, n), $(m, n-1)$, and $(m-1, n-1)$. These relationships, which exist under the assumptions of Corollary 5.1, are given by (5.6), (5.7), (5.11), and (5.12) (see also [7]-[9]).

Remark 2. From (5.17), it follows from equating coefficients of degree $m+n-1$ that

$$e^*_m q_{n-1} = f^*_0 r_0 \tag{5.19}$$

and

$$q^*_{n-1} e_m = r^*_0 f_0. \tag{5.20}$$

Thus, if the conditions of Corollary 5.1 are satisfied, then e^*_m, q_{n-1}, q^*_{n-1}, and e_m are all nonsingular. Normalizing (2.7), (2.8), (4.1), and (4.2) by setting

$$r_0 = r^*_0 = e_m = e^*_m = I, \tag{5.21}$$

rather than by (5.5), we obtain

$$H_{m,n} \cdot [q_{n-1}, \cdots, q_0]^t = [0, \cdots, 0, I]^t, \tag{5.22}$$

$$[q^*_{n-1}, \cdots, q^*_0] \cdot H_{m,n} = [0, \cdots, 0, I], \tag{5.23}$$

$$H_{m,n} \cdot [f_{n-1}, \cdots, f_0]^t = [I, 0, \cdots, 0]^t, \tag{5.24}$$

$$[f^*_{n-1}, \cdots, f^*_0] \cdot H_{m,n} = [I, 0, \cdots, 0]. \tag{5.25}$$

These conditions, together with the requirement that $\det(q_{n-1}) \neq 0$ and $\det(q^*_{n-1}) \neq 0$, are exactly the conditions given by Gohberg and Heinig [16] in deriving the inverse formula (5.4). Because of the different normalization requirement, their formula includes the term q^{-1}_{n-1} between the first two matrices and q^{*-1}_{n-1} between the last two matrices. This is permissible because of (5.19)-(5.21). In the scalar case, this is the well-known formula of Gohberg and Semencul [14].

Remark 3. The assumptions of Corollary 5.1, which are equivalent to conditions (5.22)-(5.25) of Gohberg and Heinig, are far more restrictive than the assumptions of Theorem 3.1, which are equivalent to (3.8), (3.9) and the block Yule-Walker equations (3.10) and (3.11). The formula of Gohberg and Heinig has the additional requirement that q_{n-1} and q_{n-1}^* be nonsingular (which is equivalent to $H_{m,n-1}$ being nonsingular). Thus, for example, (3.1) can be used to obtain the inverse of

$$H_{2,3} = \begin{bmatrix} I & 0 & 0 \\ 0 & 0 & I \\ 0 & I & 0 \end{bmatrix}, \tag{5.26}$$

whereas, (5.4) cannot be applied.

Remark 4. Since the assumptions of Corollary 5.1 require that not only $H_{m,n}$ but also $H_{m,n-1}$ be nonsingular, it should be possible to express the inverse of $H_{m,n-1}$ in closed form as well. Indeed, by deriving Frobenius-type identities similar to (5.6), (5.7), (5.11), and (5.12) (cf. Bultheel [7]-[9]), the matrix Padé form of type $(m-1, n-2)$ can be expressed in terms of matrix Padé forms of type $(m, n-1)$ and $(m-1, n-1)$. Then, substituting the Padé forms of type $(m, n-1)$ and $(m-1, n-2)$ into (3.1) (with n replaced by $n-1$) and simplifying, we obtain as another corollary to Theorem 3.1 the second inverse formula of Gohberg and Heinig, namely,

$$H_{m,n-1}^{-1} = \begin{bmatrix} f_{n-2} & \cdots & f_0 \\ \vdots & \iddots & \\ f_0 & & \end{bmatrix} \begin{bmatrix} q_{n-1}^* & \cdots & q_1^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} - \begin{bmatrix} q_{n-2} & \cdots & q_0 \\ \vdots & \iddots & \\ q_0 & & \end{bmatrix} \begin{bmatrix} f_{n-1}^* & \cdots & f_1^* \\ & \ddots & \vdots \\ & & f_{n-1}^* \end{bmatrix}. \tag{5.27}$$

Here, we have again normalized according to (5.5). We also note that the Gohberg-Heinig formulae given here are both determined from (3.1). Additional formulae, based on (3.2) rather than (3.1), can also be derived.

Remark 5. Gohberg and Heinig prove their formulae with the coefficients over a noncommutative algebra. Our formulae and results also carry over with minor alterations. In particular, Theorem 2.3 and Corollary 5.1 would both require that (2.9) be equivalent to (2.10) and (2.11), simultaneously.

6. The inverse formulae of Gohberg–Krupnik. Let $(L_{m-2}(z), M_{n-2}(z), N(z))$ and $(L_{m-2}^*(z), M_{n-2}^*(z), N^*(z))$ be an RMPFo and an LMPFo, respectively, of type $(m-2, n-2)$ for $A(z)$. These matrix Padé forms then satisfy

$$A(z)M_{n-2}(z) - L_{m-2}(z) = z^{m+n-3}N(z), \tag{6.1}$$

$$M_{n-2}^*(z)A(z) - L_{m-2}^*(z) = z^{m+n-3}N^*(z). \tag{6.2}$$

The inverse of $H_{m,n}$ in terms of matrix Padé forms of types $(m-2, n-2)$ and $(m-1, n-1)$ is given by Corollary 6.1.

COROLLARY 6.1. *Let the matrix Padé forms identified by* (2.7), (2.8), (6.1), *and* (6.2) *be given. Then, the following statements are equivalent*:

$$\det(H_{m,n}) \neq 0 \quad \textit{and} \quad \det(H_{m-1,n-1}) \neq 0, \tag{6.3}$$

$$\det(n_0) \neq 0, \quad \det(q_0) \neq 0, \quad \textit{and} \quad \det(r_0) \neq 0, \tag{6.4}$$

$$\det(n_0^*) \neq 0, \quad \det(q_0^*) \neq 0 \quad \textit{and} \quad \det(r_0^*) \neq 0. \tag{6.5}$$

In addition, if any (and therefore all) of the conditions (6.3), (6.4), *or* (6.5) *are satisfied,*

then

$$
\begin{aligned}
H_{m,n}^{-1} = &\begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} q_0^{-1} \begin{bmatrix} m_{n-2}^* & \cdots & m_{-1}^* \\ & \ddots & \vdots \\ & & m_{n-2}^* \end{bmatrix} \\
&- \begin{bmatrix} m_{n-3} & \cdots & m_0 & 0 & 0 \\ \vdots & \iddots & & & \\ m_0 & \iddots & & & \\ 0 & & & & \\ 0 & & & & \end{bmatrix} q_0^{*-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\
&+ \begin{bmatrix} q_{n-1} \\ \vdots \\ q_0 \end{bmatrix} q_0^{-1} [q_{n-1}^*, \cdots, q_0^*].
\end{aligned} \tag{6.6}
$$

Here, the matrix Padé forms have been normalized so that[4]

$$n_0 = n_0^* = r_0 = r_0^* = I. \tag{6.7}$$

Proof. To prove that (6.3) is equivalent to (6.4), it follows directly from Theorem 2.3 that $\det(H_{m,n}) \neq 0$ implies that $\det(r_0) \neq 0$, while $\det(H_{m-1,n-1}) \neq 0$ implies that $\det(n_0) \neq 0$ and $\det(q_0) \neq 0$. Conversely, suppose that (6.4) holds. Again, from Theorem 2.3, we have that $\det(n_0) \neq 0$ and $\det(q_0) \neq 0$ implies $\det(H_{m-1,n-1}) \neq 0$. But, then

$$
\begin{bmatrix} a_{m-n+1} & \cdots & a_m \\ \vdots & & \vdots \\ a_m & \cdots & a_{m+n-1} \end{bmatrix} \begin{bmatrix} I & & & q_{n-1} \\ & \ddots & & \vdots \\ & & I & q_1 \\ 0 & \cdots & 0 & q_0 \end{bmatrix} = \begin{bmatrix} a_{m-n+1} & \cdots & a_{m-1} & 0 \\ \vdots & & \vdots & \vdots \\ a_{m-1} & \cdots & a_{m+n-2} & 0 \\ a_m & \cdots & a_{m+n-1} & r_0 \end{bmatrix}, \tag{6.8}
$$

together with the assumption that $\det(r_0) \neq 0$, implies that also $\det(H_{m,n}) \neq 0$.

A similar argument shows that (6.3) is equivalent to (6.5).

To prove (6.6), we first establish some identities. Observe that, under the normalization condition (6.7), $(L_{m-2}(z), M_{n-2}(z), N(z))$, $(L_{m-2}^*(z), M_{n-2}^*(z), N^*(z))$, $(P_{m-1}(z)q_0^{-1}, Q_{n-1}(z)q_0^{-1}, R(z)q_0^{-1})$, and $(q_0^{*-1}P_{m-1}^*(z), q_0^{*-1}Q_{n-1}^*(z), q_0^{*-1}R^*(z))$ satisfy the conditions of Lemma 2.5, with (m, n) replaced by $(m-1, n-1)$. Here, (2.25) becomes

$$
\begin{bmatrix} Q_{n-1}(z)q_0^{-1} & M_{n-2}(z) \\ z^2R(z)q_0^{-1} & N(z) \end{bmatrix} \begin{bmatrix} N^*(z) & -M_{n-2}^*(z) \\ -z^2q_0^{*-1}R^*(z) & q_0^{*-1}Q_{n-1}^*(z) \end{bmatrix} = \begin{bmatrix} I & 0 \\ 0 & I \end{bmatrix} \tag{6.9}
$$

and, in particular,

$$\{R(z)q_0^{-1}\}N^*(z) = N(z)\{q_0^{*-1}R^*(z)\}. \tag{6.10}$$

Note that the constant and linear terms in (6.10) yield

$$q_0 = q_0^* \tag{6.11}$$

and

$$q_0^{*-1}(n_1^* - r_1^*) = (n_1 - r_1)q_0^{-1}. \tag{6.12}$$

[4] Rather than normalizing with $r_0 = r_0^* = I$, it is equally proper to normalize with $q_0 = q_0^* = I$.

For later purposes, also observe the identity

$$\begin{bmatrix} q_{n-1} & \cdots & q_0 \\ \vdots & \iddots & \\ q_0 & & \end{bmatrix} q_0^{-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} - \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} q_0^{*-1} \begin{bmatrix} 0 & q_{n-1}^* \cdots & q_1^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \\ & & 0 \end{bmatrix}$$

$$= \begin{bmatrix} q_{n-1} \\ \vdots \\ q_0 \end{bmatrix} q_0^{-1} [q_{n-1}^*, \cdots, q_0^*], \tag{6.13}$$

which follows using (6.11).

Next, we proceed as in Corollary 5.1 by constructing right and left matrix Padé forms of type (m, n) for $A(z)$. Set

$$U_m(z) = \{P_{m-1}(z)[I + (n_1 - r_1)z] - L_{m-2}(z)z^2\}q_0^{-1} \tag{6.14}$$

and

$$V_n(z) = \{Q_{n-1}(z)[I + (n_1 - r_1)z] - M_{n-2}(z)z^2\}q_0^{-1}. \tag{6.15}$$

Then, $U_m(z)$ and $V_n(z)$ provide an RMPFo of type (m, n) for $A(z)$. To see this, note that the degree requirements are clearly satisfied. In addition, the columns of $V_n(z)$ are linearly independent since, in (6.15), $v_0 = I$. Finally,

$$\begin{aligned} A(z)V_n(z) - U_m(z) &= \{[A(z)Q_{n-1}(z) - P_{m-1}(z)][I + (n_1 - r_1)z] \\ &\quad - z^2[A(z)M_{n-2}(z) - L_{m-2}(z)]\}q_0^{-1} \\ &= \{z^{m+n-1}R(z)[I + (n_1 - r_1)z] - z^{m+n-1}N(z)\}q_0^{-1} \qquad (6.16) \\ &= z^{m+n-1}\{(r_0 - n_0) + (r_1 + r_0(n_1 - r_1) - n_1)z + z^2\{\cdots\}\}q_0^{-1} \\ &= z^{m+n+1}\{\cdots\}q_0^{-1}, \end{aligned}$$

since $n_0 = r_0 = I$.

Similarly, it can be shown that

$$U_m^*(z) = q_0^{*-1}\{[I + (n_1^* - r_1^*)z]P_{m-1}^*(z) - L_{m-2}^*(z)z^2\}, \tag{6.17}$$

$$V_n^*(z) = q_0^{*-1}\{[I + (n_1^* - r_1^*)z]Q_{n-1}^*(z) - M_{n-2}^*(z)z^2\} \tag{6.18}$$

provides an LMPFo of type (m, n) for $A(z)$.

Note that (6.15) and (6.18), respectively, yield

$$\begin{bmatrix} v_{n-1} & \cdots & v_0 \\ \vdots & \iddots & \\ v_0 & & \end{bmatrix} = \begin{bmatrix} q_{n-1} & \cdots & q_0 \\ \vdots & \iddots & \\ q_0 & & \end{bmatrix} q_0^{-1} + \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix} (n_1 - r_1)q_0^{-1}$$

$$- \begin{bmatrix} m_{n-3} & \cdots & m_0 & 0 & 0 \\ \vdots & \iddots & & & \\ m_0 & \iddots & & & \\ 0 & & & & \\ 0 & & & & \end{bmatrix} q_0^{-1} \tag{6.19}$$

and

$$\begin{bmatrix} v_n^* & \cdots & v_1^* \\ & \ddots & \vdots \\ & & v_n^* \end{bmatrix} = q_0^{*-1} \begin{bmatrix} 0 & q_{n-1}^* & \cdots & q_1^* \\ & \ddots & \ddots & \vdots \\ & & & q_{n-1}^* \\ & & & 0 \end{bmatrix} + q_0^{*-1}(n_1^* - r_1^*) \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix}$$

$$- q_0^{*-1} \begin{bmatrix} m_{n-2}^* & \cdots & m_{-1}^* \\ & \ddots & \vdots \\ & & m_{n-2}^* \end{bmatrix} \tag{6.20}$$

where $m_{-1}^* = 0$. Substituting (6.19) and (6.20) into (3.1), and rearranging terms, we obtain

$$\begin{aligned} H_{m,n}^{-1} = & \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} q_0^{-1} \begin{bmatrix} m_{n-2}^* & \cdots & m_{-1}^* \\ & \ddots & \vdots \\ & & m_{n-2}^* \end{bmatrix} \\ & - \begin{bmatrix} m_{n-3} & \cdots & m_0 & 0 & 0 \\ \vdots & \iddots & & & \\ m_0 & & & & \\ 0 & & & & \\ 0 & & & & \end{bmatrix} q_0^{*-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\ & + \begin{bmatrix} q_{n-1} & \cdots & q_0 \\ \vdots & \iddots & \\ q_0 & & \end{bmatrix} q_0^{-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\ & - \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} q_0^{*-1} \begin{bmatrix} 0 & q_{n-1}^* & \cdots & q_1^* \\ & \ddots & \ddots & \vdots \\ & & & q_{n-1}^* \\ & & & 0 \end{bmatrix} \\ & + \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & \iddots & \\ q_0 & & & \\ 0 & & & \end{bmatrix} \{(n_1 - r_q) q_0^{-1} - q_0^{*-1}(n_1^* - r_1^*)\} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \end{aligned} \tag{6.21}$$

But, using (6.12) and (6.13), it is easy to see that (6.21) is exactly (6.6). □

Remark 1. The inverse formula (6.6) can also be determined by bordering techniques. Indeed, (6.8) can be further manipulated to obtain

$$H_{m,n}^{-1} = \begin{bmatrix} H_{m-1,n-1}^{-1} & 0 \\ 0 \cdots & 0 \end{bmatrix} + \begin{bmatrix} q_{n-1} \\ \vdots \\ q_0 \end{bmatrix} q_0^{-1} [q_{n-1}^*, \cdots, q_0^*]. \tag{6.22}$$

Equation (3.1) applied to $H_{m-1,n-1}^{-1}$, along with simplification using Lemma 2.5, converts (6.22) to (6.6).

The present proof takes its cue from the approach of §§ 4 and 5. In each case, the inverse formula is obtained from (3.1) using Frobenius-type identities for matrix

Padé forms. The Frobenius-type identities (6.14), (6.15), (6.17), and (6.18) used here can be found in [8] (see also [22]).

Remark 2. Note that, if the matrix Padé forms (2.7), (2.8), (6.1), and (6.2) satisfy the conditions of Corollary 6.1 and are normalized according to (6.7), then

$$H_{m,n}\cdot\left\{\begin{bmatrix} m_{n-2}\\ \vdots\\ m_0\\ 0\end{bmatrix}-\begin{bmatrix} q_{n-1}\\ \vdots\\ q_0\end{bmatrix}\cdot n_1\right\}=\begin{bmatrix} 0\\ \vdots\\ 0\\ I\\ n_1\end{bmatrix}-\begin{bmatrix} 0\\ \vdots\\ \\ 0\\ I\end{bmatrix}n_1=\begin{bmatrix} 0\\ \vdots\\ 0\\ I\\ 0\end{bmatrix}. \tag{6.23}$$

Thus, the second last column of $H_{m,n}^{-1}$ is a combination of the coefficients of $M_{n-2}(z)$ and $Q_{n-1}(z)$. Similarly, we can obtain the second last row of $H_{m,n}^{-1}$ as a combination of the coefficients of $M_{n-2}^*(z)$ and $Q_{n-1}^*(z)$.

Conversely, suppose $X=[x_{n-1},\cdots,x_0]^t$ and $Q=[q_{n-1},\cdots,q_0]^t$, respectively, represent the second last and last block columns of the inverse of $H_{m,n}$. Then, if $\det(q_0)\neq 0$, we have that

$$H_{m,n}\left\{\begin{bmatrix} x_{n-1}\\ \vdots\\ x_0\end{bmatrix}-\begin{bmatrix} q_{n-1}\\ \vdots\\ q_0\end{bmatrix}q_0^{-1}x_0\right\}=\begin{bmatrix} 0\\ \vdots\\ 0\\ I\\ -q_0^{-1}x_0\end{bmatrix} \tag{6.24}$$

so that

$$H_{m-1,n-1}\left\{\begin{bmatrix} x_{n-1}\\ \vdots\\ x_1\end{bmatrix}-\begin{bmatrix} q_{n-1}\\ \vdots\\ q_1\end{bmatrix}q_0^{-1}x_0\right\}=\begin{bmatrix} 0\\ \vdots\\ 0\\ I\end{bmatrix}. \tag{6.25}$$

This implies that

$$M_{n-2}(z)=z^{-1}\{X(z)-Q_{n-1}(z)q_0^{-1}x_0\} \tag{6.26}$$

is an RMPFo denominator of type $(m-2,n-2)$ for $A(z)$. Similarly, we can obtain an LMPFo denominator of type $(m-2,n-2)$ when we have the last and second last block rows of the inverse of $H_{m,n}$. Then substitution into (6.6) yields

$$\begin{aligned} H_{m,n}^{-1}=&\begin{bmatrix} q_{n-2}&\cdots&q_0&0\\ \vdots&\iddots&&\\ q_0&&&\\ 0&&&\end{bmatrix}q_0^{-1}\begin{bmatrix} x_{n-1}^*&\cdots&x_0^*\\ &\ddots&\vdots\\ &&x_{n-1}^*\end{bmatrix}\\ &-\begin{bmatrix} x_{n-2}&\cdots&x_0&0\\ \vdots&\iddots&&\\ x_0&&&\\ 0&&&\end{bmatrix}q_0^{*-1}\begin{bmatrix} q_{n-1}^*&\cdots&q_0^*\\ &\ddots&\vdots\\ &&q_{n-1}^*\end{bmatrix}\\ &+\begin{bmatrix} q_{n-2}&\cdots&q_0&0\\ \vdots&\iddots&&\\ q_0&&&\\ 0&&&\end{bmatrix}q_0^{-1}(x_0^*-x_0)q_0^{*-1}\begin{bmatrix} q_{n-1}^*&\cdots&q_0^*\\ &\ddots&\vdots\\ &&q_{n-1}^*\end{bmatrix} \end{aligned} \tag{6.27}$$

$$+\begin{bmatrix} q_{n-1} \\ \vdots \\ q_0 \end{bmatrix} q_0^{-1}[q_{n-1}^*, \cdots, q_0^*];$$

whereas, substitution into (3.1) gives

$$H_{m-1,n-1}^{-1} = \begin{bmatrix} q_{n-2} & \cdots & q_0 \\ \vdots & \iddots & \\ q_0 & & \end{bmatrix} q_0^{-1} \begin{bmatrix} x_{n-1}^* & \cdots & x_1^* \\ & \ddots & \vdots \\ & & x_{n-1}^* \end{bmatrix} - \begin{bmatrix} x_{n-2} & \cdots & x_0 \\ \vdots & \iddots & \\ x_0 & & \end{bmatrix} q_0^{*-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_1^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix}. \tag{6.28}$$

Remark 3. In the scalar case, if $X = [x_{n-1}, \cdots, x_0]^t$ and $Q = [q_{n-1}, \cdots, q_0]^t$ represent the second last and last columns of the inverse of $H_{m,n}$ respectively, and $q_0 \neq 0$, then (6.27) and (6.28) reduce to

$$H_{m,n}^{-1} = q_0^{-1} \left\{ \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} \begin{bmatrix} x_{n-1} & \cdots & x_0 \\ & \ddots & \vdots \\ & & x_{n-1} \end{bmatrix} \right. \tag{6.29}$$

$$\left. - \begin{bmatrix} x_{n-2} & \cdots & x_0 & 0 \\ \vdots & \iddots & \iddots & \\ x_0 & & & \\ 0 & & & \end{bmatrix} \begin{bmatrix} q_{n-1} & \cdots & q_0 \\ & \ddots & \vdots \\ & & q_{n-1} \end{bmatrix} + \begin{bmatrix} q_{n-1}^2 & \cdots & q_{n-1}q_0 \\ \vdots & & \vdots \\ q_{n-1}q_0 & \cdots & q_0^2 \end{bmatrix} \right\}$$

and

$$H_{m-1,n-1}^{-1} = q_0^{-1} \left\{ \begin{bmatrix} q_{n-2} & \cdots & q_0 \\ \vdots & \iddots & \\ q_0 & & \end{bmatrix} \begin{bmatrix} x_{n-1} & \cdots & x_1 \\ & \ddots & \vdots \\ & & x_{n-1} \end{bmatrix} \right. \tag{6.30}$$

$$\left. - \begin{bmatrix} x_{n-2} & \cdots & x_0 \\ \vdots & \iddots & \\ x_0 & & \end{bmatrix} \times \begin{bmatrix} q_{n-1} & \cdots & q_1 \\ & \ddots & \vdots \\ & & q_{n-1} \end{bmatrix} \right\}.$$

These are the original formulae of Gohberg and Krupnik [15].

Remark 4. Following the approach of § 4, we can also obtain conditions and inverse formulae for $H_{m-1,n-1}$ and $H_{m,n}$ when the first and second block column, along with the first and second block row, of the inverse of $H_{m,n}$ is given (cf. Iohvidov [19]). Here, conditions and inverse formulae for $H_{m-1,n-1}$ and $H_{m,n}$ are stated in terms of matrix Padé forms of type $(m, n-1)$ and $(m+1, n-2)$. Additional formulae, based on (3.2) rather than (3.1), can also be given.

7. The inverse formulae of Ben-Artzi and Shalom. As mentioned in § 3, the assumptions of Theorem 3.1 can be equivalently replaced by the requirement that we obtain solutions to

$$H_{m,n} \cdot [q_{n-1}, \cdots, q_0]^t = [0, \cdots, 0, I]^t, \tag{7.1}$$

$$[q_{n-1}^*, \cdots, q_0^*] \cdot H_{m,n} = [0, \cdots, 0, I], \tag{7.2}$$

$$H_{m,n} \cdot [v_n, \cdots, v_1]^t = -[a_{m+1}, \cdots, a_{m+n-1}, a_{m+n}]^t, \tag{7.3}$$

$$[v_n^*, \cdots, v_1^*] \cdot H_{m,n} = -[a_{m+1}, \cdots, a_{m+n-1}, a_{m+n}] \tag{7.4}$$

where a_{m+n} can be any $p \times p$ matrix. It is possible to alter the right-hand sides of (7.3) and (7.4) and still obtain inverse formulae for $H_{m,n}$. In particular, we may replace the right-hand sides by linear combinations of the rows and columns of $H_{m+1,n}$.

LEMMA 7.1. *Let $H_{m,n}$ be the block Hankel matrix* (1.1). *Suppose there are solutions to* (7.1) *and* (7.2) *along with solutions to*

$$H_{m,n}\cdot[x_{n-1},\cdots,x_0]^t = H_{m+1,n}\cdot[y_{n-1},\cdots,y_0]^t, \tag{7.5}$$

and

$$[x_{n-1}^*,\cdots,x_0^*]\cdot H_{m,n} = [y_{n-1}^*,\cdots,y_0^*]\cdot H_{m+1,n}, \tag{7.6}$$

with y_0 and y_0^* nonsingular. Then $H_{m,n}$ is nonsingular with inverse

$$\begin{aligned}&\begin{bmatrix} y_{n-1}-x_{n-2} & \cdots & y_1-x_0 & y_0 \\ \vdots & \iddots & & \\ y_1-x_0 & & & \\ y_0 & & & \end{bmatrix} y_0^{-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\ &+\begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} y_0^{*-1} \begin{bmatrix} x_{n-1}^* & x_{n-2}^*-y_{n-1}^* & \cdots & x_0^*-y_1^* \\ & \ddots & \ddots & \vdots \\ & & & x_{n-2}^*-y_{n-1}^* \\ & & & x_{n-1}^* \end{bmatrix},\end{aligned} \tag{7.7}$$

or, equivalently, the inverse is given by

$$\begin{aligned}&\begin{bmatrix} q_{n-1} & & \\ \vdots & \ddots & \\ q_0 & \cdots & q_{n-1} \end{bmatrix} y_0^{*-1} \begin{bmatrix} y_{n-1}^*-x_{n-2}^* & \cdots & y_1^*-x_0^* & y_0^* \\ \vdots & \iddots & & \\ y_1^*-x_0^* & & & \\ y_0^* & & & \end{bmatrix} \\ &+\begin{bmatrix} x_{n-1} & & & \\ x_{n-2}-y_{n-1} & & \ddots & \\ \vdots & \ddots & & \\ x_0-y_1 & \cdots & x_{n-2}-y_{n-1} & x_{n-1} \end{bmatrix} y_0^{-1} \begin{bmatrix} q_{n-2}^* & \cdots & q_0^* & 0 \\ \vdots & \iddots & & \\ q_0^* & & & \\ 0 & & & \end{bmatrix}.\end{aligned} \tag{7.8}$$

Proof. Since

$$\begin{aligned}\begin{bmatrix} a_{m-n+1} & \cdots & a_m \\ \vdots & & \vdots \\ a_m & \cdots & a_{m+n-1} \end{bmatrix}\cdot\begin{bmatrix} x_{n-1} \\ \vdots \\ x_0 \end{bmatrix} &= \begin{bmatrix} a_{m-n+2} & \cdots & a_{m+1} \\ \vdots & & \vdots \\ a_{m+1} & \cdots & a_{m+n} \end{bmatrix}\cdot\begin{bmatrix} y_{n-1} \\ \vdots \\ y_0 \end{bmatrix} \\ &= \begin{bmatrix} a_{m-n+1} & \cdots & a_m \\ \vdots & & \vdots \\ a_m & & a_{m+n-1} \end{bmatrix}\cdot\begin{bmatrix} 0 \\ y_{n-1} \\ \vdots \\ y_1 \end{bmatrix} \\ &\quad+\begin{bmatrix} a_{m+1} \\ \vdots \\ a_{m+n} \end{bmatrix}\cdot y_0,\end{aligned} \tag{7.9}$$

we get that

$$\begin{bmatrix} v_n \\ \vdots \\ v_1 \end{bmatrix} = \left\{\begin{bmatrix} 0 \\ y_{n-1} \\ \vdots \\ y_1 \end{bmatrix} - \begin{bmatrix} x_{n-1} \\ x_{n-2} \\ \vdots \\ x_0 \end{bmatrix}\right\}\cdot y_0^{-1} \tag{7.10}$$

is a solution to (7.3). Similarly,

$$[v_n^*, \cdots, v_1^*] = y_0^{*-1} \cdot ([0, y_{n-1}^*, \cdots, y_1^*] - [x_{n-1}^*, x_{n-2}^*, \cdots, x_0^*]) \tag{7.11}$$

is a solution to (7.4). Substituting (7.10) and (7.11) into (3.1) gives (7.7), while substituting into (3.2) gives (7.8). □

Let $E^{(i)}$ denote the $n \times 1$ block matrix having the $p \times p$ identity matrix as its ith block row, and zeros elsewhere. Similarly, let $E^{*(i)}$ be the $1 \times n$ block matrix having the $p \times p$ identity matrix as its ith block column, with zeros elsewhere. Theorem 7.2 shows how to construct the inverse of a block Hankel matrix, knowing only the last block column and row, along with two successive block columns and rows of the inverse.

THEOREM 7.2. *Let $H_{m,n}$ be the block Hankel matrix* (1.1). *Suppose there are solutions to* (7.1) *and* (7.2), *along with solutions to*

$$H_{m,n} \cdot [x_{n-1}, \cdots, x_0]^t = E^{(i)}, \tag{7.12}$$

$$H_{m,n} \cdot [y_{n-1}, \cdots, y_0]^t = E^{(i+1)}, \tag{7.13}$$

$$[x_{n-1}^*, \cdots, x_0^*] \cdot H_{m,n} = E^{*(i)}, \tag{7.14}$$

$$[y_{n-1}^*, \cdots, y_0^*] \cdot H_{m,n} = E^{*(i+1)}. \tag{7.15}$$

If, in addition, y_0 and y_0^* are both nonsingular, then $H_{m,n}$ is nonsingular with inverse given by

$$\begin{aligned} &\begin{bmatrix} y_{n-1} - x_{n-2} & \cdots & y_1 - x_0 & y_0 \\ \vdots & \iddots & & \\ y_1 - x_0 & & & \\ y_0 & & & \end{bmatrix} y_0^{-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix} \\ &+ \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} y_0^{*-1} \begin{bmatrix} x_{n-1}^* & x_{n-2}^* - y_{n-1}^* & \cdots & x_0^* - y_1^* \\ & \ddots & \ddots & \vdots \\ & & & x_{n-2}^* - y_{n-1}^* \\ & & & x_{n-1}^* \end{bmatrix} \\ &+ \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} y_0^{*-1}(y_{n-i}^* - y_{n-i}) \cdot y_0^{-1} \begin{bmatrix} q_{n-1}^* & \cdots & q_0^* \\ & \ddots & \vdots \\ & & q_{n-1}^* \end{bmatrix}; \end{aligned} \tag{7.16}$$

or, equivalently, the inverse is given by

$$\begin{aligned} &\begin{bmatrix} q_{n-1} & & \\ \vdots & \ddots & \\ q_0 & \cdots & q_{n-1} \end{bmatrix} y_0^{*-1} \begin{bmatrix} y_{n-1}^* - x_{n-2}^* & \cdots & y_1^* - x_0^* & y_0^* \\ \vdots & \iddots & & \\ y_1^* - x_0^* & & & \\ y_0^* & & & \end{bmatrix} \\ &+ \begin{bmatrix} x_{n-1} & & & \\ x_{n-2} - y_{n-1} & & & \\ \vdots & \ddots & & \\ x_0 - y_1 & \cdots & x_{n-2} - y_{n-1} & x_{n-1} \end{bmatrix} y_0^{-1} \begin{bmatrix} q_{n-2}^* & \cdots & q_0^* & 0 \\ \vdots & \iddots & & \\ q_0^* & & & \\ 0 & & & \end{bmatrix} \\ &+ \begin{bmatrix} q_{n-1} & & \\ \vdots & \ddots & \\ q_0 & \cdots & q_{n-1} \end{bmatrix} \cdot y_0^{*-1} \cdot (y_{n-i} - y_{n-i}^*) \cdot y_0^{-1} \cdot \begin{bmatrix} q_{n-2}^* & \cdots & q_0^* & 0 \\ \vdots & \iddots & & \\ q_0^* & & & \\ 0 & & & \end{bmatrix}. \end{aligned} \tag{7.17}$$

Proof. Equation (7.13) implies that

$$H_{m+1,n} \cdot Y = E^{(i)} + E^{(n)} \cdot c \tag{7.18}$$

where

$$c = a_{m+1} \cdot y_{n-1} + \cdots + a_{m+n} \cdot y_0. \tag{7.19}$$

Therefore

$$H_{m,n} \cdot (X + Q \cdot c) = H_{m+1,n} \cdot Y, \tag{7.20}$$

and similarly we can show that

$$(X^* + c^* \cdot Q^*) \cdot H_{m,n} = Y^* \cdot H_{m+1,n} \tag{7.21}$$

where

$$c^* = y^*_{n-1} \cdot a_{m+1} + \cdots + y^*_0 \cdot a_{m+n}. \tag{7.22}$$

Therefore, using Lemma 7.1, $H_{m,n}$ is nonsingular with inverse given according to (7.7) or (7.8) applied to equations (7.1), (7.2), (7.20), and (7.21). For example, substituting these expressions into (7.7) and expanding, we obtain the inverse of $H_{m,n}$ as

$$\begin{aligned}
&\begin{bmatrix} y_{n-1} - x_{n-2} & \cdots & y_1 - x_0 & y_0 \\ \vdots & \iddots & & \\ y_1 - x_0 & & & \\ y_0 & & & \end{bmatrix} \bar{y}_0^{-1} \begin{bmatrix} q^*_{n-1} & \cdots & q^*_0 \\ & \ddots & \vdots \\ & & q^*_{n-1} \end{bmatrix} \\
&+ \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} y_0^{*-1} \begin{bmatrix} x^*_{n-1} & x^*_{n-2} - y^*_{n-1} & \cdots & x^*_0 - y^*_1 \\ & \ddots & & \vdots \\ & & & x^*_{n-2} - y^*_{n-1} \\ & & & x^*_{n-1} \end{bmatrix} \\
&+ \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & & \\ q_0 & & & \\ 0 & & & \end{bmatrix} (y_0^{*-1} \cdot c^* - c \cdot y_0^{-1}) \begin{bmatrix} q^*_{n-1} & \cdots & q^*_0 \\ & \ddots & \vdots \\ & & q^*_{n-1} \end{bmatrix}.
\end{aligned} \tag{7.23}$$

To obtain formula (7.16), we note that

$$\begin{aligned}
y^*_{n-i} &= Y^* \cdot E^{(i)} \\
&= Y^* \cdot H_{m,n} \cdot X \\
&= Y^* \cdot \{H_{m+1,n} \cdot Y - H_{m,n} \cdot Q \cdot c\} \\
&= Y^* \cdot \{H_{m+1,n} \cdot Y - E^{(n)} \cdot c\} \\
&= Y^* \cdot H_{m+1,n} \cdot Y - y_0^* \cdot c \\
&= Y^* H_{m,n} \cdot [0, y_{n-1}, \cdots, y_1]^t + c^* y_0 - y_0^* \cdot c \\
&= E^{*(i+1)} \cdot [0, y_{n-1}, \cdots, y_1]^t + c^* y_0 - y_0^* \cdot c \\
&= y_{n-i} + c^* y_0 - y_0^* \cdot c.
\end{aligned} \tag{7.24}$$

Therefore

$$(y_0^{*-1} \cdot c^* - c \cdot y_0^{-1}) = y_0^{*-1} \cdot (y^*_{n-i} - y_{n-i}) \cdot y_0^{-1}. \tag{7.25}$$

Substituting (7.25) into (7.23) gives (7.16). Formula (7.17) is verified in a similar manner. □

Remark 1. In the scalar case, Theorem 7.2 gives the inverse of $H_{m,n}$ in terms of the last column along with an additional two successive columns of the inverse. In this case, (7.16) gives $H_{m,n}^{-1}$ as

$$y_0^{-1}\left\{\begin{bmatrix} y_{n-1}-x_{n-2} & \cdots & y_1-x_0 & y_0 \\ \vdots & \iddots & \iddots & \\ y_1-x_0 & \iddots & & \\ y_0 & & & \end{bmatrix}\begin{bmatrix} q_{n-1} & \cdots & q_0 \\ & \ddots & \vdots \\ & & q_{n-1} \end{bmatrix}\right. \tag{7.26}$$
$$\left. + \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & \iddots & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix}\begin{bmatrix} x_{n-1} & x_{n-2}-y_{n-1} & \cdots & x_0-y_1 \\ & \ddots & \ddots & \vdots \\ & & & x_{n-2}-y_{n-1} \\ & & & x_{n-1} \end{bmatrix}\right\}.$$

Formula (7.26) is due to Ben-Artzi and Shalom [3] (in its Hankel formulation). Equation (7.17) reduces to an alternate formula in the scalar case.

Remark 2. Let S be the $n \times n$ shift matrix having 1's along the superdiagonal, and 0's elsewhere. Suppose in the scalar case there is a solution Q, to (7.1) along with a solution to

$$H_{m,n} \cdot Z = S \cdot H_{m,n} \cdot Y \tag{7.27}$$

where $y_0 \neq 0$. Then, there is also a solution to (7.5) since

$$H_{m,n} \cdot (Z + Q \cdot c) = H_{m+1,n} \cdot Y \tag{7.28}$$

where c is given by (7.19). Since $y_0 \neq 0$, Lemma 7.1 implies that $H_{m,n}$ is nonsingular, with inverse given by (7.7). After simplification, this inverse formula is

$$y_0^{-1}\left\{\begin{bmatrix} y_{n-1}-z_{n-2} & \cdots & y_1-z_0 & y_0 \\ \vdots & \iddots & \iddots & \\ y_1-z_0 & \iddots & & \\ y_0 & & & \end{bmatrix}\begin{bmatrix} q_{n-1} & \cdots & q_0 \\ & \ddots & \vdots \\ & & q_{n-1} \end{bmatrix}\right. \tag{7.29}$$
$$\left. + \begin{bmatrix} q_{n-2} & \cdots & q_0 & 0 \\ \vdots & \iddots & \iddots & \\ q_0 & \iddots & & \\ 0 & & & \end{bmatrix}\begin{bmatrix} z_{n-1} & z_{n-2}-y_{n-1} & \cdots & z_0-y_1 \\ & \ddots & \ddots & \vdots \\ & & & z_{n-2}-y_{n-1} \\ & & & z_{n-1} \end{bmatrix}\right\}.$$

This is the main inverse formula of Ben-Artzi and Shalom [3] in the scalar case. They use this formula to give simple derivations of their own scalar formula (7.26), along with other inverse formulae including the formulae of both Gohberg-Krupnik and Gohberg-Semencul.

8. Conclusions. The Frobenius-type relationships given in this paper are but a small sample of similar recurrence relationships that exist between matrix Padé forms that have been developed by Bultheel [7]-[9]. All the relationships require the existence of inverses of certain coefficients in the Padé forms involved. These requirements are always satisfied for normal matrix power series (where $H_{m,n}$ is nonsingular for all m and n). For this restricted class of power series, many of the recursive relationships provide directly algorithms for the computation of Padé forms. Depending on the path (within the Padé table) determined by the recurrence, Bultheel observes that most previous algorithms [1], [5], [13], [23], [25]-[27], [32] that explicitly or implicitly compute the inverse of Hankel or Toeplitz matrices are equivalent to using an appropriate recurrence formula.

For a subset of these relationships, this paper shows that each recurrence yields a separate closed formula for the inverse of a block Hankel matrix. Algorithms based on recurrences that specify computations along an off-diagonal path (e.g., [1], [5], [27], [32]) yield closed formulae expressed by (3.1), (3.2), and (6.6). Those that specify computations along a staircase (e.g., [13], [23], [25]) yield formulae (5.4) and (5.27); whereas, those that specify computations along an antidiagonal path yield (4.8) and (4.9). Additional closed formulae can be derived corresponding to other recurrences given by Bultheel.

Formulae (5.4), (5.27), and (6.6) are equivalent to those given by Gohberg and Heinig and Gohberg and Krupnik, whereas (3.1), (3.2), (4.8), and (4.9) are new. A major advantage of the new formulae is that the underlying assumptions are far less restrictive than they are for (5.4), (5.27), and (6.6). Whereas, the new formulae require only that $H_{m,n}$ be nonsingular, the latter also require that an additional submatrix be nonsingular. In addition, necessary and sufficient conditions for the existence of $H_{m,n}^{-1}$ are directly available from the coefficients of Padé forms. This provides a significant computational advantage.

Relaxed conditions provide little computational gain, however, if the available algorithms can function only under the more severe restrictions of normality. Unfortunately, this is true for most algorithms that compute nonscalar Padé forms or decompose block Hankel matrices. One exception in this regard is the MPADE algorithm of Labahn and Cabay [22]. This algorithm is based on a recurrence relationship between Padé forms at successive nonsingular nodes along an off-diagonal path of the matrix Padé table (or, by reversing coefficients, along an antidiagonal path). When the power series is normal, or, less restrictively, when all principal minors of the associated Hankel matrix are nonsingular (e.g., when the block Hankel matrix is positive definite), all the nodes along the path are nonsingular and then their recurrence relationship reduces to (6.15), which is one of many given by Bultheel. The methods based on this relationship are special cases of the MPADE algorithm

For purposes of expressing the inverse of $H_{m,n}$ in terms of the new formulae (3.1), (3.2), (4.8), and (4.9), the MPADE algorithm is particularly suitable. Singularity is detected with no additional effort. When $H_{m,n}$ is nonsingular, the necessary Padé forms (i.e., the solutions of the associated block Yule-Walker equations) appearing in the formulae are simultaneously available on termination. The algorithm has no restrictions of normality. In addition, intermediate results enable the computation of the inverses of any nonsingular principal minors.

Using classical polynomial arithmetic, the cost of the MPADE algorithm is typically $O(p^3n^2)$, but can reach a complexity of $O(p^3n^3)$ in pathological cases (e.g., when all the principal minors are singular). When the power series is normal, this cost is the same as that of previously mentioned algorithms.

Using fast polynomial arithmetic in the normal case, Bitmead and Anderson [4] indicate that their scalar algorithm, based on a divide-and-conquer partitioning of the Hankel matrix, can be generalized to the nonscalar case with a cost complexity of $O(p^3n \log^2 n)$. Under somewhat relaxed normality conditions (i.e., near-normality), Labahn [21] also gives an algorithm, an adaptation of MPADE, with the same complexity.

In the scalar case, one call of an algorithm given by Cabay and Choi [11] can be used to construct the inverse formulae (3.1), (3.2), (4.8), or (4.9) with cost complexity $O(n \log^2 n)$ under no restrictions of normality. This is also true of other methods (cf. Sugiyama [29] for a survey) and, in particular, this is true of the method of Brent, Gustavson, and Yun [6]. They use two calls of a fast antidiagonal GCD algorithm,

EMGCD, to determine the two Padé forms required by the Gohberg-Semencul formula (5.4). The algorithm succeeds immediately if both $H_{m,n}$ and $H_{m,n-1}$ are nonsingular. If $H_{m,n-1}$ is singular (but $H_{m,n}$ is not), then a nonsingular matrix $H_{m,n+1}$ is first constructed (it is not clear that this is always possible in the nonscalar case). Two additional calls of the antidiagonal algorithm are then made to yield the two Padé forms required by the second formula (5.27) of Gohberg and Semencul. By computing the inverse of $H_{m,n}$ using (4.8) or (4.9), their algorithm can now be altered so as to only require one call of their antidiagonal algorithm.

The use of (3.1) to express the inverse of $H_{m,n}$ avoids the immediate problem of potential numerical instabilities inherent when using instead the two formulae (5.4) and (5.27) according to the status of singularity of relevant minors (cf. Bunch [10]). However, this does not imply that the algorithm for determining the inverse of $H_{m,n}$ using (3.1) is stable, since this first requires the stable computation of $(P(z), Q(z))$ and $(U(z), V(z))$. The question of the stability of the algorithm MPADE for computing $(P(z), Q(z))$ and $(U(z), V(z))$ is an open question currently under investigation.

Acknowledgment. We would like to thank the reviewer for bringing the paper of Ben-Artzi and Shalom to our attention.

REFERENCES

[1] H. AKAIKE, *Block Toeplitz matrix inversion*, SIAM J. Appl. Math., 24 (1973), pp. 234-241.
[2] G. S. AMMAR AND W. B. GRAGG, *The generalized Schur algorithm for the superfast solution of Toeplitz systems*, Lecture Notes in Mathematics 1237, Springer-Verlag, Berlin, New York, 1987, pp. 315-330.
[3] A. BEN-ARTZI AND T. SHALOM, *On inversion of Toeplitz and close to Toeplitz matrices*, Linear Algebra Appl., 75 (1986), pp. 173-192.
[4] R. R. BITMEAD AND B. D. O. ANDERSON, *Asymptotically fast solutions of Toeplitz and related systems of linear equations*, Linear Algebra Appl., 34 (1980), pp. 103-116.
[5] N. K. BOSE AND S. BASU, *Theory and recursive computation of* 1-D *matrix Padé approximants*, IEEE Trans. Circuits and Systems, 4 (1980), pp. 323-325.
[6] R. BRENT, F. G. GUSTAVSON, AND D. Y. Y. YUN, *Fast solution of Toeplitz systems of equations and computation of Padé approximants*, J. Algorithms, 1 (1980), pp. 259-295.
[7] A. BULTHEEL, *Recursive algorithms for the matrix Padé table*, Math. Comp., 35 (1980), pp. 875-892.
[8] ———, *Recursive relations for block Hankel and Toeplitz systems Part* I: *Direct recursions*, J. Comput. Appl. Math., 10 (1984), pp. 301-328.
[9] ———, *Recursive relations for block Hankel and Topelitz systems Part* II: *Dual recursions*, J. Comput. Appl. Math., 10 (1984), pp. 329-354.
[10] J. R. BUNCH, *Stability of methods for solving Toeplitz systems of equations*, SIAM J. Sci. Comput., 6 (1985), pp. 349-364.
[11] S. CABAY AND D. K. CHOI, *Algebraic computations of scaled Padé fractions*, SIAM J. Comput., 15 (1986), pp. 243-270.
[12] D. K. CHOI, *Algebraic computations of scaled Padé fractions*, Ph.D. thesis, University of Alberta, Edmonton, Alberta, Canada, 1984.
[13] J. DURBIN, *The fitting of time-series models*, Rev. Inst. Internat. Statist., 28 (1960), pp. 233-244.
[14] I. C. GOHBERG AND A. A. SEMENCUL, *On the inversion of finite Toeplitz matrices and their continuous analogs*, Mat. Issled., 2 (1972), pp. 201-233. (In Russian.)
[15] I. C. GOHBERG AND N. YA. KRUPNIK, *A formula for the inversion of finite Toeplitz matrices*, Mat. Issled., 2 (1972), pp. 272-283. (In Russian.)
[16] I. C. GOHBERG AND G. HEINIG, *Inversion of finite Toeplitz matrices made of elements of a non-commutative algebra*, Rev. Roumaine Math. Pures Appl., XIX(5) (1974), pp. 623-663. (In Russian.)
[17] W. B. GRAGG, *The Padé table and its relation to certain algorithms of numerical analysis*, SIAM Rev., 14 (1972), pp. 1-61.
[18] F. DE HOOG, *A new algorithm for solving Toeplitz systems of equations*, Linear Algebra Appl., 88 (1987), pp. 123-138.
[19] I. S. IOHVIDOV, *Hankel and Toeplitz Matrices and Forms*, Birkhauser, Boston, 1982.

[20] T. KAILATH, S.-Y. KUNG, AND M. MORF, *Displacement ranks of matrices and linear equations*, J. Math. Anal. Appl., 68 (1979), pp. 395–407.

[21] G. LABAHN, *Matrix Padé approximants*, M.Sc. thesis, Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada, 1986.

[22] G. LABAHN AND S. CABAY, *Matrix Padé fractions and their computation*, SIAM J. Comput., 18 (1989), pp. 639–657.

[23] N. LEVINSON, *The Wiener* RMS *(root mean square) error in filter design*, J. Math. Phys., 25 (1947), pp. 261–278.

[24] R. J. MCELIECE AND J. B. SHEARER, *A property of Euclid's Algorithm and an application to Padé approximation*, SIAM J. Appl. Math., 34 (1978), pp. 611–617.

[25] M. MORF, A. VIEIRA, AND D. T. LEE, *Ladder forms for identification and speech processing*, in Proc. Conference on Decision and Control, December 7–9, 1977.

[26] B. R. MUSICUS. *Levinson and fast Choleski algorithms for Toeplitz and quasi-Toeplitz matrices*, Laboratory of Electronics, Massachusetts Institute of Technology, Cambridge, MA, 1984.

[27] J. RISSANEN, *Algorithms for triangular decomposition of block Hankel and Toeplitz matrices with application to factoring positive matrix polynomials*, Math. Comp., 27 (1973), pp. 147–154.

[28] ———, *Solution of linear equations with Hankel and Toeplitz matrices*, Numer. Math., 22 (1974), pp. 361–366.

[29] Y. SUGIYAMA, *An algorithm for solving discrete-time Wiener–Hopf equations based on Euclid's Algorithm*, IEEE Trans. Inform. Theory, 32 (1986), pp. 394–409.

[30] W. F. TRENCH, *An algorithm for the inversion of finite Hankel matrices*, SIAM J. Appl. Math., 13 (1965), pp. 1102–1107.

[31] G. A. WATSON, *An algorithm for the inversion of block matrices of Toeplitz form*, J. Assoc. Comput. Mach., 20 (1973), pp. 409–415.

[32] R. A. WIGGENS AND E. A. ROBINSON, *Recursive solution to the multichannel filtering problem*, J. Geophys. Res., 70 (1965), pp. 1885–1891.

[33] S. ZOHAR, *Toeplitz matrix inversion: The algorithm of W. F. Trench*, J. Assoc. Comput. Mach., 16 (1969), pp. 592–601.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 124-132, February 1990

007

A DENSITY THEOREM FOR PURELY ITERATIVE ZERO FINDING METHODS*

JOEL FRIEDMAN†

Abstract. In this paper a wide class of purely iterative root finding methods is proved to work for all complex valued polynomials with a positive probability depending only on the method and the degree of the polynomial. More precisely, if the set of polynomials with roots in the unit ball is considered, then for fixed degree the area of convergent points in the ball of radius 2 is bounded below by some constant for any purely iterative method $z_{i+1} \leftarrow T_f(z_i)$, where $T_f(z)$ is a rational function of z and f_i and its derivatives, for which (1) ∞ is repelling fixed point for all f of degree greater than 1 and (2) $T_f(z)$ depends only on z and f's roots and commutes with linear maps on the complex plane.

Key words. root finding, iterative methods, Newton's method, polynomials

AMS(MOS) subject classification. 65H05

1. Introduction. The goal of this paper is to prove a theorem about the density of points for which a purely iterative root finding method converges to a root.

For $z \in \mathbf{C}$ and $f(z) = \sum_{i=0}^{d} a_i z^i$ consider a map

$$T_f(z) = \frac{P(z, f, f', \cdots, f^{(l)})}{Q(z, f, f', \cdots, f^{(l)})}$$

where P and Q are polynomials over $\mathbf{C}$. For each f, T_f is a map from $\mathbf{C} \cup \{\infty\}$ to itself which we think of as an iteration in a root finding method. We require that

(1)

$$T_f(z) = \frac{z^s P_0(f, zf', z^2 f'', \cdots)}{z^{s-1} Q_0(f, zf', z^2 f'', \cdots)} \tag{1.1}$$

where P_0 and Q_0 are homogeneous polynomials of the same degree.

(2) $T_f(z)$ depends only on z and the roots $r_1, \cdots, r_d$ of f, and

$$A(T_f(z)) = T_{Af}(Az)$$

for any linear map $A: z \mapsto az + b$, where

$$Af(z) = a_d(z - Ar_1) \cdots (z - Ar_d)$$

for

$$f(z) = a_d(z - r_1) \cdots (z - r_d).$$

(3) $T_f(r) = r$, $|T_f'(r)| < 1$ for any root r of f.

(4) $T_f(\infty) = \infty$, $|T_f'(\infty)| > 1$ for any f of degree greater than 1.

To measure the density of convergent points for T_f, let P_d denote the polynomials of degree d with roots in the unit ball. For a polynomial f, let

$$\Gamma_{T,f} = \{z : T_f^n(z) \to \text{a root of } f \text{ as } n \to \infty\}$$

where T_f^n is the nth iterate of T_f (i.e., $\Gamma_{T,f}$ is the set of points converging to a root of f under the iteration T_f). Let

$$A_{T,f} = |\Gamma_{T,f} \cap B_2(0)|.$$

Then $A_{T,f}/4\pi$ is the probability that a random point in $B_2(0)$ converges to a root.

* Received by the editors November 30, 1987; accepted for publication (in revised form) March 7, 1989.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08544 (jf@princeton.edu).

THEOREM 1.1. *Let T satisfy* (1)-(4). *Then for any d there is a $c>0$ such that*

$$A_{T,f} > c \qquad \forall f \in P_d.$$

Furthermore, we have

$$A_{T,f,r} > c \qquad \forall f \in P_d, \qquad \forall r \text{ with } f(r)=0,$$

where $A_{T,f,r}$ denotes the contribution to $A_{T,f}$ from the root r of f.

More precisely,

$$A_{T,f,r} = |\Gamma_{T,f,r} \cap B_2(0)|$$

where

$$\Gamma_{T,f,r} = \{z : T_f^n(z) \to r \text{ as } n \to \infty\}.$$

The above density theorem was conjectured to hold for Newton's method by Smale in [Sma85]. This conjecture was proven in [Fri86]; the proof used some special properties of Newton's method and explicit bounds on the constants as a function of d were given. The above theorem applies to a much larger class of root finding methods, though no explicit bounds on c are given.

Examples of T satisfying (1)-(4) are

(1) Newton's method, $T_f(z) = z - (f/f')$.

(2) Modified Newton's method, $T_f(z) = z - h(f/f')$ with a constant h, $0<h<1$.

(3) Taylor's method (see [Atk78])

$$T_f(z) = z + \sum_{i=1}^{k} \frac{d^i}{dt^i}\left(\frac{\phi_t(z)}{i!}\right)\Bigg|_{t=0} h^i$$

where $\phi_t(z)$ solves

$$\frac{d\phi_t(z)}{dt} = -\frac{f(z)}{f'(z)}, \qquad \phi_0(z) = z$$

with k a positive integer and h a positive number sufficiently small (depending on k).

(4) Incremental Euler's method (see [Atk78])

$$T_f(z) = z + \sum_{i=1}^{k} \frac{(-hf(z))^k}{k!} g^{(k)}(f(z))$$

with $g = f^{-1}$, k a positive integer, and h positive and sufficiently small.

(5) Any iterate of a T satisfying (1)-(4). This shows that maps satisfying (1)-(4) may contain extraneous attractive fixed points. For example, Newton's method, even applied to polynomials of degree as low as three, can contain attractive periodic points of period two. Therefore the second iterate of Newton's method can have extraneous attractive fixed points.

To prove Theorem 1.1, take any sequence $f_m \in P_d$; we will show that A_{T,f_n} cannot approach 0 as n tends to infinity. By passing to a subsequence we can assume each of the f_n's coefficients, or equivalently each of f_n's roots, converge. If $A_{T,f}$ were continuous in f at the limit of the f_n's, then we would be done; the fact that $A_{T,f}$ can be discontinuous at f's having multiple roots makes the theorem more interesting. If the limit of f_n has at least one isolated root, one would get enough of a contribution to A_{T,f_n} from such an isolated root (for large n) to show that A_{T,f_n} is bounded away from zero. However, if all of f_n's roots tend to cluster into several groups as $n \to \infty$, we must look at the limiting geometry of each individual cluster to estimate A_{T,f_n}. So fix a cluster, and "blow-up" the picture of the roots at that cluster so that, while they

remain in some bounded region, they separate into smaller subclusters. Again, if the blow-up of at least one cluster has as least one isolated root, we are done; the reason is that the isolated root's contribution to Γ_{T,f_n} contains a sequence of balls that, from the point of view of the subcluster, tend to ∞ and whose radii get larger. These balls, from the point of view of the original scale of the problem, look like a sequence of balls whose radii get smaller and smaller and whose center converges to the cluster's limit point. It is not hard to see that in the original scale of the problem, the largest ball is of appreciable size, thus bounding A_{T,f_n} from below.

If none of the clusters has an isolated root, we look at the geometry of each subcluster, blowing up the picture at each subcluster. Since the blowing-up process separates a cluster of roots into at least two distinct subclusters, successive blowing-up eventually isolates the roots. One can then find balls in Γ_{T,f_n} for each root in the blow-up of the picture that isolates it, and back up through the blow-ups until reaching the original scale of the problem, finding balls in each scale of blow-up that lie in Γ_{T,f_n}. The fact that this can be done for each root proves, in addition, the second part of the theorem.

The basic estimate for the existence of the aforementioned balls is Lemma 2.2, proved in § 2. In § 3 we describe the blowing-up process more precisely and show how Lemma 2.2 can be applied backwards through the blowing-up process.

2. Some preliminary results. Let $g\colon N \to \mathbf{C} \cup \{\infty\}$ be a complex analytic map, for an open $N \subset \mathbf{C} \cup \{\infty\}$. Let $z \in N$ be a repelling fixed point, i.e., $g(z) = z$ and $|g'(z)| > 1$.

LEMMA 2.1. *For any open $A \subset \mathbf{C}$ we have that for n sufficiently large,*

$$g^n\{B_\varepsilon(z)\} \cap A \neq \varnothing.$$

Proof. Apply Cauchy's formula for $(fg^n)'(z)$, where f is a Möbius function taking a point in A to ∞, and where the contour is a small circle around z.

For our maps T, we have that ∞ is a repelling fixed point so the lemma can be applied.

From condition (1)-(4) on T it is easy to see that

$$T_f'(\infty) = q(d) = \frac{Q_0(1, d, d(d-1), \cdots)}{P_0(1, d, d(d-1), \cdots)}$$

is a rational function of d independent of f, and that if r is a k-tuple root, then

$$T_f'(r) = \frac{P_0(1, k, k(k-1), \cdots)}{Q_0(1, k, k(k-1), \cdots)} = \frac{1}{q(k)}.$$

For any f we have that for z in a neighborhood of ∞,

$$T_f(z) = \frac{z}{q(d)} + O(1)$$

and

$$T_f'(z) = \frac{1}{q(d)} + O\left(\frac{1}{|z|}\right) \tag{2.1}$$

and T_f^{-1} is defined locally. We have

$$\frac{T_f(z)}{z} = \frac{1}{q(d)} + O\left(\frac{1}{|z|}\right)$$

and so for $|z|$ sufficiently large, we have $z_0 = z$, z_{-1}, z_{-2}, $\cdots$ given by $T_f(z_{-i}) = z_{-i+1}$ has $|z_{-n}|$ growing like $(q(d)-\varepsilon)^n$ for any $\varepsilon > 0$ depending on how large $|z|$ is, and thus

$$\frac{z-n}{z} = \prod_{i=0}^{n-1}\left(1 - O\left(\frac{1}{|z_{-i}|}\right)\right)q(d) = q^n(d)\left(1 - O\left(\frac{1}{|z|}\right)\right). \tag{2.2}$$

The mean value theorem and (2.1) yield for, say, $r < |z|/2$,

$$T_f^n\{B_{r'}(z_{-n})\} \subset B_r(z) \tag{2.3}$$

with

$$r' = rq^n(d)\left(1 - O\left(\frac{1}{|z_{-n+1}|} + \cdots + \frac{1}{|z|}\right)\right) = rq^n(d)\left(1 - O\left(\frac{1}{|z|}\right)\right). \tag{2.4}$$

Let

$$\tilde{z} = \lim_{n\to\infty} \frac{z_{-n}}{q^n(d)},$$

the limit existing by virtue of (2.2). For any $r < |z|/2$, using (2.3) and (2.4), we have

$$T_f^n\{B_{rq^n(d)/2}(\tilde{z}q^n(d))\} \subset B_r(z) \tag{2.5}$$

for n sufficiently large (depending on r).

Next we would like to obtain a version of (2.5) for polynomials close to f in a certain sense. Fix d, D, and f, and consider the set $\mathcal{F}_{f,\delta,D}$ of polynomials

$$g(z) = (z - s_1)\cdots(z - s_{d+D})$$

with $s_i \in B_\delta(r_i)$ for $1 \le i \le d$ and $|s_i| > 1/\delta$ for $i > d$.

LEMMA 2.2. *For any sufficiently large z and $r < |z|/2$ there is a c, δ_0, and n_0 such that if $\delta < \delta_0$ and $n > n_0$ we have*

$$T_g^n\{B_{rq^n(d)/2}(\tilde{z}q^n(d))\} \subset B_r(z)$$

if

$$|\tilde{z}|q^n(d) < \frac{c}{\delta}$$

for all $g \in \mathcal{F}_{f,\delta,D}$.

Proof. Dividing both numerator and denominator by $z^{s-1}g^{\deg(P_0)}$ in condition (1) on T yields

$$T_g(z) = \frac{zP_0(1, z(g'/g), z^2(g''/g), \cdots)}{Q_0(1, z(g'/g), z^2(g''/g), \cdots)}.$$

For $|z|$ sufficiently large and, say, less than or equal to $1/2\delta$ we have

$$\left|\frac{f'}{f} - \frac{g'}{g}\right| \le \sum_{i=1}^{d}\left|\frac{1}{z-r_i} - \frac{1}{z-s_i}\right| + \sum_{j=d+1}^{d+D}\left|\frac{1}{z-s_j}\right|$$

$$= \sum\left|\frac{s_i - r_i}{(z-r_i)(z-s_i)}\right| + \sum\frac{1}{|z-s_i|} = O\left(\frac{\delta}{|z|^2} + \delta\right).$$

Similarly, we have

$$\left|\frac{f^{(k)}}{f}-\frac{g^{(k)}}{g}\right| \leqq \sum_{1\leqq i_1<\cdots<i_k\leqq d} k!\left|\frac{1}{(z-r_{i_1})\cdots(z-r_{i_k})}-\frac{1}{(z-s_{i_1})\cdots(z-s_{i_k})}\right|$$
$$+\sum_{1\leqq i_1<\cdots<i_k\leqq d+D, i_k>d} k!\left|\frac{1}{(z-s_{i_1})\cdots(z-s_{i_k})}\right|$$
$$=O\left(\frac{\delta}{|z|^{k+1}}+\frac{\delta}{|z|^{k-1}}+\frac{\delta^2}{|z|^{k-2}}+\cdots+\delta^k\right)=O\left(\frac{\delta}{|z|^{k+1}}+\frac{\delta}{|z|^{k-1}}\right);$$

in the last line we have used the fact that for sufficiently large z and n we have $|z|\delta<1$ (which follows from the second equation in the statement of the lemma). Thus

$$\left|z^k\frac{f^{(k)}}{f}-z^k\frac{g^{(k)}}{g}\right|=O\left(\frac{\delta}{|z|}+\delta|z|\right)$$

and so

$$T_g(z)=T_f(z)\left(1+O\left(\frac{\delta}{|z|}+\delta|z|\right)\right),$$
$$T_g'(z)=T_f'(z)\left(1+O\left(\frac{\delta}{|z|}+\delta|z|\right)\right). \tag{2.6}$$

We caution the reader that the big-O notation above is as the quantity in parenthesis tends to zero and that the constants in the big-O notation depend on d and D. Now fix a z sufficiently large and a small ε so that $z_0=z$, z_{-1}, z_{-2}, $\cdots$ defined as before grow like a geometric series. Then, using (2.6), we see that for δ sufficiently small we have that $y_0=z$, y_{-1}, y_{-2}, $\cdots$, y_{-n} given by $T_g(y_{-i})=y_{-i+1}$ grows like a geometric series, as long as $|y^{-n}|<c/\delta$ for c sufficiently small. Then we get

$$y_{-n}=z_{-n}\prod_{i=0}^{n-1}\left(1+O\left(\frac{\delta}{|y_{-i}|}+\delta|y_{-i}|\right)\right)=z_{-n}\left(1+O\left(\frac{\delta}{|z|}+\delta|y_{-n}|\right)\right).$$

Using the chain rule, we have

$$(T_g^n)'(w)=\prod_{i=0}^{n-1}T_g'(T_g^i(w))=\left(\frac{1}{q(d)}\right)^n\left(1+O\left(\frac{\delta}{|T_g^n(w)|}+\delta|w|\right)\right)$$

assuming $|T_g^n(w)|$ is sufficiently large and $|w|\leqq c/\delta$. The mean value theorem then implies

$$T_g^n\{B_{r'}(z_{-n})\}\subset B_r(z)$$

where

$$r'=rq^n(d)\left(1+O\left(\frac{\delta}{|z|}+\delta|z_{-n}|\right)\right).$$

Hence, as before, we get that for sufficiently large n,

$$T_g^n\{B_{rq^n(d)/2}(\tilde{z}q^n(d))\}\subset B_r(z)$$

as long as $|\tilde{z}|q^n(d)<c/\delta$ for c sufficiently small.

3. Successive normalizations. The difficulty in proving Theorem 1.1 is that $A_{T,f}$ is not necessarily continuous when f has multiple roots. Let $f_1, f_2, \cdots$ be a sequence in P_d, and $r_1^1, r_1^2, \cdots$ a sequence of respective roots for which

$$\lim_{n\to\infty} A_{T,f_n,r_1^n} = \inf_{f\in P_d, f(r)=0} A_{T,f,r}.$$

By passing to a subsequence we may assume that

$$f_n(z) = (z-r_1^n)^{e_1}\cdots(z-r_{k_0}^n)^{e_{k_0}}$$

with

$$e_1+\cdots+e_{k_0} = d$$

and

$$r_i^n \neq r_j^n \qquad \forall n, \qquad i<j\leqq k_0.$$

By passing to a subsequence we can assume

$$r_i^n \to r_i \quad \text{as } n\to\infty.$$

If r_1 is isolated, i.e., $e_1 = 1$, then it would be easy to show that for some $\delta>0$ we have

$$B_\delta(r_i^n)\subset \Gamma_{T,f_n}$$

for all n sufficiently large, and thus

$$\inf_{f\in P_d, f(r)=0} A_{T,f,r} > 0$$

(the details of the argument appear as part of the proof later in this section). If not, we can assume

$$r_1 = r_2 = \cdots = r_{k_1}$$

and $r_j \neq r_1$ for $j>k_1$. We will now analyze more carefully the way in which $r_1^n, \cdots, r_{k_1}^n$ converge to r_1.

For $z_1, \cdots, z_m \in \mathbf{C}$ not all the same, we define the *normalization* of $z_1, \cdots, z_m$ *centered at* z_1 to be the unique linear map

$$g(z) = az+b, \qquad a\in\mathbf{R},\ a>0,\ b\in\mathbf{C}$$

such that

$$\sum_{i<j} |g(z_i)-g(z_j)| = 1,$$

and $g(z_1)=0$.

By passing to a subsequence we can assume that

(1) the normalizations of $r_1^n, \cdots r_{k_0}^n$, $g_n(z) = a_n^z + b_n$, centered at r_1^n have $g_n(r_i^n)\to s_i$ as $n\to\infty$, and

(2)

$$q_1^{\lfloor -\log_{q_1} a_n\rfloor} a_n \to a \tag{3.1}$$

as $n\to\infty$ for some $a\in[1/q_1, 1]$, where

$$q_1 = q\left(\sum_{i=1}^{k_1} e_i\right)$$

and where $\lfloor\beta\rfloor$ denotes the largest integer $\leqq\beta$.

Clearly

$$\sum_{i<j} |s_i - s_j| = 1,$$

and so we have

$$s_1 = \cdots = s_{k_2}$$

and $s_j \neq s_1$ for $j > k_2$, where $k_2 < k_1$. In other words, by normalizing we separate the first k_1 roots into smaller groups. By repeated normalization we will finally separate r_1^n from all other r_i^n's. Now we start with the deepest level of normalization and work up, proving a density lower bound for each level.

Let the deepest level be ℓ, and let

$$h_n(r_i^n) \to t_i \qquad \text{for } 1 \leqq i \leqq k_\ell$$

where h_n is the normalization of $r_1^n, \cdots, r_{k_\ell}^n$ centered at r_1^n. We have

$$\sum_{i<j} |t_i - t_j| = 1,$$

$t_1 = 0$, and $t_i \neq t_1$ if $i > 1$. Consider

$$\tilde{f}(z) = (z - t_1)^{e_1} \cdots (z - t_{k_\ell})^{e_{k_\ell}}.$$

Since $T_{\tilde{f}}(t_1) = t_1$, $|T'_{\tilde{f}}(t_1)| < 1$, and ∞ is a repelling fixed point for $T_{\tilde{f}}$, we have open sets E, arbitrarily near ∞, such that $T_{\tilde{f}}^n\{E\} \to t_1$ as $n \to \infty$. Take a point z large enough so that Lemma 2.2 holds, with $B_\varepsilon(z)$ converging to t_1 uniformly under $T_{\tilde{f}}$ for some $\varepsilon > 0$ (we can assure uniform convergence by assumption (3) of § 1). We have

$$B_{\varepsilon q_\ell^m/2}(\tilde{z} q_\ell^m) \subset \Gamma_{T,\tilde{f}}$$

for m sufficiently large, where $\tilde{z}$ is as in Lemma 2.2 and

$$q_\ell = q\left(\sum_{i=1}^{k_\ell} e_i\right).$$

Let h'_n be the normalization of the $\ell - 1$th level, i.e., of $r_1^n, \cdots, r_{k_{\ell-1}}^n$ centered at r_1^n,

$$h'_n(z) = a'_n z + b'_n$$

and let

$$h_n(z) = a_n z + b_n.$$

We have that

$$\frac{a_n}{a'_n} q_\ell^{\lfloor -\log_{q_\ell}(a_n/a'_n) \rfloor} \to a$$

as $n \to \infty$ for some $a \in [1/q_\ell, 1]$ (at each level we normalize and pass to a subsequence satisfying a condition analogous to that of (3.1) as well as the preceeding condition). We want to prove that

$$B_{\varepsilon_0}(z_0) \subset \Gamma_{T,h'_n f_n}, \tag{3.2}$$

for all sufficiently large n, where

$$z_0 = \tilde{z} a q_\ell^{-M},$$

$$\varepsilon_0 = \varepsilon a q_\ell^{-M}/4,$$

for some positive integer M; this will complete the first stage moving backwards through the normalizations, each time finding a ball of fixed size with respect to the current normalization in Γ_{T,f_n} for sufficiently large n. To prove (3.2) first consider

$$h_n f_n(z) = (z - h_n(r_1^n))^{e_1} \cdots (z - h_n(r_{k_\ell}^n))^{e_{k_\ell}}.$$

We claim that for n sufficiently large we have

$$B_\varepsilon(z) \subset \Gamma_{T,h_n f_n}.$$

To see this, we note that for some small $\eta > 0$ we have

$$|z - t_1| \leqq \eta \Rightarrow |T_{\tilde{f}}(z) - t_1| \leqq (1-\mu)|z - t_1|,$$

by assumption (3) of § 1, for some $\mu > 0$, and that for some large N,

$$T_f^N\{B_\varepsilon(z)\} \subset B_{\eta/2}(t_1)$$

by the uniform convergence. Estimating as in Lemma 2.2 (note that for any δ we have $h_n f_n \in \mathcal{F}_{\tilde{f},\delta,D}$ for n sufficiently large and $D = d - q_\ell$) we get that for n sufficiently large

$$|z - t_1| \leqq \eta \Rightarrow |T_{h_n f_n}(z) - t_1| \leqq (1 - \mu/2)|z - t_1| \Rightarrow z \in \Gamma_{T,h_n f_n}$$

and that

$$T_{h_n f_n}^N\{B_\varepsilon(z)\} \subset B_\eta(t_1) \subset \Gamma_{T,h_n f_n}$$

using $h_n(r_1^n) = t_1$ and that for any $y \in B_\varepsilon(z)$ we have y, $T_{\tilde{f}}(y)$, $T_{\tilde{f}}^2(y), \cdots$ stays away from the r_i^n's with $i > 1$. Now we apply Lemma 2.2 to conclude that for m sufficiently large we have

$$T_{h_n f_n}^m\{B_{\varepsilon q_\ell^m/2}(\tilde{z} q_\ell^m)\} \subset B_\varepsilon(z) \subset \Gamma_{T,h_n f_n}$$

so that

$$B_{\varepsilon q_\ell^m/2}(\tilde{z} q_\ell^m) \subset \Gamma_{T,h_n f_n}$$

as long as $|\tilde{z}| q_\ell^m < c/\delta$ for some c sufficiently small, where $1/\delta$ is a lower bound on $h_n(r_i^n)$ for $i > k_\ell$. Rescaling by a factor of a_n/a_n' and translating appropriately we get

$$B_{\varepsilon q_\ell^m a_n/(2a_n')}(\tilde{z} q_\ell^m a_n/a_n') \subset \Gamma_{T,h_n' f_n}$$

if

$$|\tilde{z}| q_\ell^m a_n/a_n' < c \min_{i > k_\ell} h_n'(r_i^n) < c'. \tag{3.3}$$

Taking

$$m(n) = \left\lfloor \log_{q_\ell} \frac{a_n'}{a_n} \right\rfloor - M,$$

where M is sufficiently large to ensure (3.3) holds, we get that for sufficiently large n,

$$B_{\varepsilon a q_\ell^{-M}/4}(\tilde{z} a q_\ell^{-M}) \subset \Gamma_{T,h_n' f},$$

the 4 in $\varepsilon a q_\ell^{-M}/4$ appearing to account for the fact that

$$\frac{a_n}{a_n'} q_\ell^{m(n)}$$

approaches, rather than equals, $a q_\ell^{-M}$ as $n \to \infty$. Thus (3.2) is established.

Now that we have a statement of the form

$$B_{\varepsilon_0}(z_0) \subset \Gamma_{T,h_n' f_n},$$

we proceed to get a statement of the form

$$B_{\varepsilon_1}(z_1) \subset \Gamma_{T,h_n'' f_n},$$

where h_n'' is the normalization at the $\ell-2$th level, i.e., the normalization of $r_1^n, \cdots, r_{k_{\ell-2}}^n$ centered at z_1^n. To do this, we consider

$$\hat{f}(z) = (z - t_1)^{e_1} \cdots (z - t_{k_{\ell-1}})^{e_{k_{\ell-1}}}.$$

Using Lemma 2.1 we can find an arbitrarily large z with an ε so that for some N

$$T_{\hat{f}}^N\{B_{\varepsilon/2}(z)\} \subset B_{\varepsilon_0}(z_0).$$

Now we repeat the argument of before to conclude

$$T_{h_n' f_n}^N\{B_\varepsilon(z)\} \subset B_{\varepsilon_0}(z_0),$$

i.e.,

$$B_\varepsilon(z) \subset \Gamma_{T, h_n' f_n}$$

(with uniform convergence) for n sufficiently large, and that

$$T_{h_n'' f_n}^{m'(n)}\{B_{\varepsilon_1}(z_1)\} \subset \Gamma_{T, h_n'' f_n}$$

(again with uniform convergence) for some $m'(n)$ and fixed ε_1, z_1.

Repeating the above argument $\ell-2$ more times yields that for all n sufficiently large we have

$$B_\varepsilon(z) \subset \Gamma_{T, f_n}$$

for some fixed ε and z with z very near r_1^n. Hence

$$\lim_{n \to \infty} A_{T, f_n, r_1^n} > \pi\varepsilon^2 > 0$$

and Theorem 1.1 is proven.

REFERENCES

[Atk78] K. ATKINSON, *An Introduction to Numerical Analysis*, John Wiley, New York, 1978.

[Fri86] J. FRIEDMAN, *On the convergence of Newton's method*, in Proc. 27th Annual Symposium on Foundations of Computer Science, 1986, pp. 153-161.

[Sma85] S. SMALE, *On the efficiency of algorithms of analysis*, Bull. Amer. Math. Soc., 13 (1985), pp. 87-121.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 133–142, February 1990

008

A FAST PARALLEL HORNER ALGORITHM*

MICHAEL L. DOWLING†

Abstract. The simple Horner algorithm solves the problem of evaluating a polynomial of degree d with n indeterminates; in this paper it is shown that its implementation on a parallel computer with $O(d)$ processors can achieve a complexity of $2\lceil\log_2(d+1)\rceil\cdot(\lceil\log_2 n\rceil+1)$. If, in addition, the evaluation of all partial derivatives is also sought, then the full Horner algorithm solves this problem on a parallel computer with $O(d^{n+1})$ processors, achieving a parallel complexity of $2\lceil\log_2(d+1)\rceil\cdot(\lceil\log_2(d+1)\rceil+\lceil\log_2 n\rceil+1)$.

Key words. parallel algorithms, algebraic complexity, parallel polynomial evaluation

AMS(MOS) subject classifications. 68C25, 68C05, 68B10

1. Introduction. This article applies a technique for parallelising sequential programs to the problem of polynomial evaluation. Given a program implementation of the Horner algorithm, it is shown that sufficient information can be obtained from its semantics for the program to be reconstructed with a high degree of parallelisability. The reconstruction process has two phases, the first of which is to find optimal, parallel hyperplanes in the loops. The possible values that the index variables can take for a given loop constitute a subset A of integral n-space $\mathbf{N}^n$, where n is the number of nestings. An optimal hyperplane is an hyperplane in $\mathbf{N}^n$ containing no data dependencies, the details of which are given below, and that has a minimal number of translates that contain at least one element of A. Once such an hyperplane H has been found, the loop is reorganised so that each iteration of the outer most loop corresponds to iterating over all those indices in A belonging to a translate of H. The result of this phase is merely to reorder the sequence in which the iterations are performed, but so that all the inner loops can be executed simultaneously, for each possible value of the index variable in the outer loop. Since the same operations are being performed, the result of the first phase has no effect on the numerical stability of the algorithm. Moreover, the balance between the number of additions and multiplications enjoyed by the Horner algorithm is preserved, thereby making the transformed code well suited to execution on a vector computer with separate and independent functional units for addition and multiplication.

The second phase is to represent the values of the various iterations of the code, after the hyperplane transformation has been performed, as the solution of a lower triagonal, linear system of equations. One then applies a variant of the standard, numerical, cyclic reduction algorithm to solve this system in logarithmic time. Since the technique used in this paper operates primarily on program code, it has much wider applicability than merely to polynomial evaluation. With the Horner algorithm, however, the linear system of equations that one obtains is known explicitly, so that solutions can be computed very quickly.

Hitherto, the main problem with using vector and parallel computers was that the Horner algorithm is difficult to parallelise. As a result, much work has been devoted to finding completely new, parallel algorithms (cf. [2, p. 162]). Although such algorithms generally achieve logarithmic complexity, it is not usually possible to implement them

* Received by the editors February 16, 1988; accepted for publication (in revised form) April 11, 1989.
† Abteilung für Mathematische Optimierung, Institut für Angewandte Mathematik, Carolo-Wilhelmina Universität zu Braunschweig, Pockelsstrasse 14, D-3300 Braunschweig, Federal Republic of Germany (I1041301@DBSTU1.BITNET).

efficiently on extant computer hardware. In contrast, the first phase of the parallelisation process presented in this paper is very effective for vector computers, while further benefit can also be obtained by implementing the second phase for vector computers with several processors.

Since parallel algorithms are constructed from sequential ones, the objective of this paper is similar to that in [6], where it was shown that, if a sequential algorithm requires k operations, then there is a parallel version that requires $O(\log_2(d) \cdot \log_2(k))$ parallel steps. That result was improved by Valiant et al. in [14], where it was shown that the same complexity can be achieved with the use of $O((kd)^{\alpha})$ processors for some constant α, as opposed to the $O(k^{\log_2 d})$ required by Hyafil. The most efficient lower bound known is $\max\{\log_2 d, \log_2 k\}$ (cf. [2]).

This paper is organised as follows. Section 2 introduces the method of hyperplane parallelisation for the univariate, full Horner algorithm. Here the degree of program loop nesting is only two, so that hyperplanes are merely straight lines, thereby making the basic technique readily comprehensible without introducing unnecessarily complicated terminology. It is this section that also introduces the notion of flow dependence developed by Banerjee in [1]. The idea of using hyperplanes to parallelise sequential code originated from [11], and was developed further in [3]. Having whole hyperplanes of iterations execute concurrently is essential to applying the cyclic reduction technique introduced in § 3, which begins with the univariate, simple Horner algorithm, where the application of cyclic reduction is particularly direct. This section ends by applying the same procedure to the full Horner algorithm after hyperplane parallelisation has already been applied. The remaining two sections apply these techniques to the bivariate case. Unfortunately, Horner evaluation for polynomials with n indeterminates requires n nested DO-loops; the idea of treating the bivariate case explicitly while only alluding to the general case is therefore designed to simplify otherwise excessively complicated notation. Proofs of the statements concerning arbitrary numbers of indeterminates can readily be supplied using induction.

2. The hyperplane parallelisation. Recall that if $p(x) = a_0 + a_1 x + \cdots + a_d x^d$ is a polynomial function of the single variable x over the real number field, then the simple Horner algorithm evaluates p at a point x according to the bracketing scheme:

$$p(x) = a_0 + (a_1 + \cdots + (a_{d-2} + (a_{d-1} + a_d \cdot x) \cdot x) \ \cdots) \cdot x.$$

This can be expressed as a recurrence formula:

$$b_{d+1} = 0 \quad \text{(initialisation)},$$

$$b_j = a_j + b_{j+1} \cdot x, \qquad j = d, \cdots, 0.$$

The full Horner algorithm also evaluates all the derivatives $p^{(i)}(x)$ at the point x, for $i = 0, \cdots, d$, and amounts to a d-fold repetition of the simple Horner algorithm. The resulting recurrence formulae are given below:

$$\left.\begin{array}{ll} b_j^{(-1)} = a_j & \text{for } j = 0, 1, \cdots, d \\ b_{d+1}^{(i)} = 0 & \text{for } i = -1, 0, \cdots, d \end{array}\right\} \quad \text{(initialisation)}$$

and

$$b_j^{(i)} = b_j^{(i-1)} + b_{j+1}^{(i)} \cdot x \quad \text{where } j = d, d-1, \cdots, i, \text{ and } i = 0, 1, \cdots, d.$$

Ultimately, $i \,|\, b_i^{(i)} = p^{(i)}(x)$, where $p^{(i)}(x)$ denotes the ith derivatives of the polynomial p, and $i = 0, 1, \cdots, d$; details can be found in [4]. Figure 1 depicts a possible FORTRAN implementation, where it is presumed that the array B has already been initialised.

```
      DO 1, I = 0, D
        DO 1, J = D-1, I, -1
1         B(I,J) = B(I-1,J)+X*B(I,J+1)
```

FIG. 1. *A naïve Horner code.*

Remark. Since the dependencies amongst the data determine the parallelisability of a code segment, it is necessary to include an explicit implementation here. If, for example, the array B(I,J) above were to be coded as B(J), the code would still be correct. However, the price for memory optimisation quite often is diminished performance as a parallel algorithm, as in the case in point (cf. [3]).The more redundancy there is in the representation of the data in a parallel computer, the more ways there are of addressing them without risk to the data's integrity.

The loop in Fig. 1 is clearly not amenable to parallel execution as each iteration requires the values of the previous iterations in order to proceed. It can, however, be restructured for concurrent execution in $2(d+1)$ parallel steps. To show this, the notion of data dependence is required.

DEFINITION. Two statements, s and t, are said to be *flow dependent* if s executes before t, and t uses a value computed during the execution of s.

In [1], Banerjee introduced three notions of data dependence, one of which is that of flow dependence. Since the other two do not occur in the above code, they shall not be discussed here. The key interest in data dependencies results from a theorem, due to Banerjee, where it is shown that if the statements of a block of code were to be permuted in a manner respecting the order of execution of dependent statements, then both the original and the permuted code will always produce the same output if given the same input. In particular, where there are no data dependencies present, the order of execution is immaterial, so that there is no obstacle to concurrent execution. These data dependencies have been exploited to good practical advantage (cf. Kuck et al. [10]), and also provide a means of treating parallelisation problems theoretically (cf. [3]).

If one considers the graph whose nodes correspond to the various iterates of a loop such as that shown in Fig. 1, and whose directed edges correspond to data dependencies, then the graph corresponding to the Horner code has nodes labelled by pairs (i,j), where $i=0,1,\cdots,d$ and $i\leqq j\leqq d$, and where there are flow dependence edges from nodes $(i-1,j)$ to (i,j), and also from $(i,j+1)$ to (i,j), whenever these ordered pairs correspond to nodes. The graph corresponding to a polynomial of degree five is illustrated in Fig. 2.

From this diagram, it is obvious why the implementation of the Horner code above does not admit concurrent execution; the nested loop executes down each column in turn, from left to right. As such, the loop iterates *precisely* along lines of data dependencies, so that sequential execution is obligatory. By executing along the diagonal lines, $i-j=\text{const.}$, the obstruction to parallel execution vanishes. In this simple, two-dimensional case, these lines constitute what more generally shall be called *optimal hyperlines.* Transforming the loop so that these diagonal lines are parallel to the j-axis, say via the unimodular transformation defined by setting $k=j-i$ and $l=j$, circumvents the problem. The resulting code is therefore parallelised. It can readily be seen that this transformation is indeed optimal in the sense of minimising the number of parallel steps. The resulting code is given in Fig. 3.

Note that now the inner DO-loop has been completely freed of data dependencies so that, for each value of the outer index K, the entire inner loop can be computed in

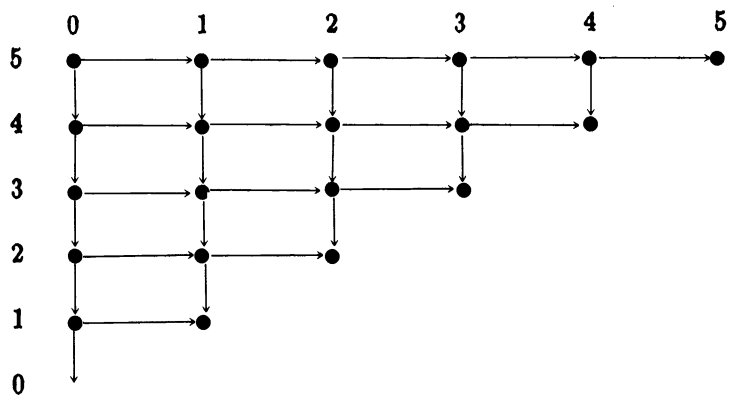


FIG. 2. *The dependence graph of the full Horner code.*

two parallel steps. The first step performs the multiplication while the second uses the result of the multiplication to compute the sum, and this for *all* possible values of L, for the current value of K. Execution is therefore completed in $2(d+1)$ parallel steps, whereas at most $d+1$ processing elements are required. This revised code is exactly equivalent to the original, so that it even yields the same, numerically insignificant digits as the naïve version. It shall soon be seen that this basic idea can be applied to multivariate polynomials to show that the full multivariate Horner algorithm also can be programmed with merely linear parallel complexity.

```
      DO 1, K = D, 0, -1
        DO 1, L = K, D-1
1         B(L-K,L)=B(L-K-1,L) + X*B(L-K, L+1)
```

FIG. 3. *A revised Horner code.*

3. A logarithmic reduction for univariate polynomials. The linear recurrence formula for the simple, univariate Horner algorithm amounts to solving the bidiagonal linear system of equations in Fig. 4.

The obvious solution is to observe that $b_d = a_d$, and $b_i = a_i + x \cdot b_{i+1}$, for each $i < d$. This is the method used in both the code segments above, but, as before, each iteration depends upon the value computed during the previous one, thereby precluding any parallel processing. The solution is to apply the cyclic reduction algorithm for solving tridiagonal systems in logarithmic time (cf. [8], [13] for details). For bidiagonal systems, cyclic reduction uses each odd numbered row to eliminate the off-diagonal entry in the even numbered row immediately below it. The result of this single parallel step is indicated in Fig. 5.

$$\begin{pmatrix} 1 & & & & & \\ -x & 1 & & & & \\ & -x & 1 & & & \\ & & -x & 1 & & \\ & & & \ddots & \ddots & \\ & & & & -x & 1 \end{pmatrix} \cdot \begin{pmatrix} b_d \\ b_{d-1} \\ b_{d-2} \\ \vdots \\ b_0 \end{pmatrix} = \begin{pmatrix} a_d \\ a_{d-1} \\ a_{d-2} \\ \vdots \\ a_0 \end{pmatrix}$$

FIG. 4. *The simple Horner linear system.*

$$\begin{pmatrix} 1 & & & & & & & \\ 0 & 1 & & & & & & \\ & -x & 1 & & & & & \\ & -x^2 & 0 & 1 & & & & \\ & & & -x & 1 & & & \\ & & & -x^2 & 0 & \ddots & 1 & \ddots \\ & & & & \ddots & \ddots & \ddots & \end{pmatrix} \cdot \begin{pmatrix} b_d \\ b_{d-1} \\ b_{d-2} \\ b_{d-3} \\ b_{d-4} \\ b_{d-5} \\ \vdots \end{pmatrix} = \begin{pmatrix} a_d \\ a_{d-1}+xa_d \\ a_{d-2} \\ a_{d-3}+xa_{d-2} \\ a_{d-4} \\ a_{d-5}+xa_{d-4} \\ \vdots \end{pmatrix}$$

FIG. 5. *The first step of cyclic reduction.*

Note that now each odd numbered variable can be computed in a single, parallel step once the even numbered variables are known. These, on the other hand, are decoupled from the odd numbered variables, and satisfy a bidiagonal system of equations of half the original size. Repeating the process, one readily sees that $4\lceil \log_2 (d+1) \rceil$ parallel steps are required for the process to terminate, the factor of four corresponding to the simultaneous multiplications and subsequent subtractions, and to the fact that $\lceil \log_2 (d+1) \rceil$ iterations are required to reduce the system of linear equations to the trivial system containing only a single unknown. The same number of iterations are required for the subsequent substitutions, so that the total number of iterations is $2\lceil \log_2 (d+1) \rceil$, each requiring two parallel steps. Moreover, the steps requiring the most processors are the first and the last, both of which require $\lfloor (d+1)/2 \rfloor$.

Remark. The application of cyclic reduction above is essentially the binary splitting algorithm of Dorn (cf. [2, p. 132]).

One notes that, since powers of x accumulate in the off-diagonal entries as the computation progresses, the cyclic reduction version of the Horner algorithm is only stable for $|x| \leqq 1$. In contrast, the standard procedure will produce better results whenever the coefficients decrease sufficiently rapidly as the degree increases. On the other hand, where $|x| < 1$, it is not always necessary to continue the recursion until the bidiagonal system has been reduced to a scalar equation. Once x^n has been reduced to a value smaller than the rounding errors, the bidiagonal system may be regarded as being diagonal, and so soluble in a single, parallel step.

The revised code for the full Horner algorithm executes the whole lines, $j - i = k$, concurrently, while the input data required for each iteration are computed during the $(k-1)$st iteration. The result is essentially a bidiagonal system, but with vector rather than scalar unknowns, so that cyclic reduction can now be applied. More precisely, the revised code can be written in FORTRAN-8X style as follows:

```
      DO 1, K = D, 0, -1
1     B(*-K,*) = B(*-(K+1),*) + X*B((*+1)-(K+1), (*+1)),
```

where B(*-K,*) corresponds to the $(d-k+1)$-dimensional vector B(L-K,L), and where L=K,$\cdots$,D. (B(-1,K) has the value a_k.) Let $\mathbf{b}^{(k)}$ and $\mathbf{a}^{(k)}$ denote the $(d+2)$-dimensional vectors

$$\mathbf{b}_l^{(k)} = \begin{cases} \text{the value of B(L,K+L)} & \text{if } 0 \leqq l \leqq d-k, \\ 0 & \text{if } d-k+1 \leqq l \leqq d, \end{cases} \quad \text{and}$$

$$\mathbf{a}_l^{(k)} = \begin{cases} a_k & \text{if } l = 0, \\ 0 & \text{otherwise.} \end{cases}$$

Here, k and l denote the values of K and L, respectively. The code segment above corresponds to the following vector recurrence formula

$$\mathbf{b}_l^{(k)} = \mathbf{b}_{l-1}^{(k+1)} + x\mathbf{b}_l^{(k+1)} + \mathbf{a}^{(k)}, \qquad k = d, d-1, \cdots, 0, \quad l = 1, \cdots, d-k,$$

which in turn can be written in matrix form as in Fig. 6 where I denotes the $(d+1)\times(d+1)$ identity matrix, and S is the shift operator, given by $S(\mathbf{v})_i = \mathbf{v}_{i+1}$ for $i < d+1$, and $S(\mathbf{v}_{d+2}) = 0$. Also, $\mathbf{a}^{(k)}$ is the vector whose sole, nonzero component is $\mathbf{a}_0^{(k)} = a_k$. In particular, $(i|\mathbf{b}_0)_i = p^{(i)}(x)$, the ith derivative of p.

$$\begin{pmatrix} I & & & & \\ -(S+xI) & I & & & \\ & -(S+xI) & I & & \\ & & \ddots & \ddots & \\ & & & -(S+xI) & I \end{pmatrix} \cdot \begin{pmatrix} \mathbf{b}^{(d)} \\ \mathbf{b}^{(d-1)} \\ \mathbf{b}^{(d-2)} \\ \vdots \\ \mathbf{b}^{(0)} \end{pmatrix} = \begin{pmatrix} \mathbf{a}^{(d)} \\ \mathbf{a}^{(d-1)} \\ \mathbf{a}^{(d-2)} \\ \vdots \\ \mathbf{a}^{(0)} \end{pmatrix}$$

FIG. 6. *A bidiagonal system for the full algorithm.*

Applying cyclic reduction blockwise, the number of blocks is halved during each reduction, so that the reduction phase terminates after $\lceil \log_2 (d+1) \rceil$ iterations. During the kth reduction, the matrix $-(S+xI)^{2^k}$ accumulates in the subdiagonal blocks. This is the lower triagonal matrix whose rth lower subdiagonal contains the rth term in the expansion of $(x+1)^{2^k}$. All of these matrices can be computed in a single, parallel step from the currently computed entries. The number of nonzero entries in the right-hand side vector blocks doubles during each iteration so that during the kth iteration, $(S+xI)^{2^k}\mathbf{a}^{(2^{k-1})} + \mathbf{a}^{(2^k)}$ requires 2^k additions, and hence k parallel steps using cyclic doubling.

The second phase of the block, cyclic reduction algorithm entails the back substitution of $-(S+xI)^{-2^{k-1}}(\mathbf{a}^{(2^k)})$ in the 2^{k-1} blocks; a process that again involves 2^k additions, and hence k parallel steps. Note that the matrix entries of $-(S+xI)^{-2^k}$ are known, namely,

$$-(S+xI)_{ij}^r = \begin{cases} (-1)^{i-j+1}\binom{r+i-j-1}{i-j-1} x^{r+j-i} & \text{if } i \geqq j, \\ 0 & \text{otherwise.} \end{cases}$$

This follows easily from the well-known formula,

$$\binom{a}{b} = \binom{a-1}{b-1} + \binom{a-2}{b-2} + \cdots + \binom{b-1}{b-1}.$$

The resulting parallel complexity for the full, univariate Horner Algorithm is, therefore,

$$2(1+2+\cdots+\lceil \log_2 (d+1) \rceil) = \lceil \log_2 (d+1) \rceil (\lceil \log_2 (d+1) \rceil + 1)$$

where the factor of two results from considering both the phases required by cyclic reduction. Note that the maximal number of processors is $\lfloor (d+1)/2 \rfloor$. During the first step, $\lfloor (d+1)/2 \rfloor$ additions and multiplications are performed, one for every second block. Thereafter, the number of blocks is halved during each step, while the number of additions doubles, until $2^{\lceil (d+1)/2 \rceil}$ additions are performed in a single block. (Identical blocks do not have to be computed more than once.)

4. The simple, multivariate, Horner algorithm. Evaluation of polynomials in several variables hinges on the fact that any element $f \in \mathbf{R}[x_1, \cdots, x_n]$ can be regarded as an element of $\mathbf{R}[x_1, \cdots, x_{n-1}][x_n]$, the ring of all polynomials in the single indeterminate, x_n, with coefficients in the polynomial ring $\mathbf{R}[x_1, \cdots, x_{n-1}]$. The preceding discussion of the univariate case therefore starts an inductive procedure for evaluating multivariate polynomials. Moreover, this applies equally to the simple algorithm as to the full Horner algorithm, which additionally computes all the partial derivatives.

The simple bivariate Horner algorithm for evaluating the polynomial

$$p(x, y) = \sum_{i+j=0}^{d} a_{ij} x^i y^j,$$

now corresponds to the linear recurrence formulae:

$$\left.\begin{aligned} &b_{ij} = 0 \quad \text{for } i+j = d+1, \\ &b_{ij} = a_{ij} + x \cdot b_{i+1j} \quad \text{for } i = d-j, d-j-1, \cdots, 1, \\ &b_{0j} = b_{0j} + x \cdot b_{1j} + y \cdot b_{0j+1}, \end{aligned}\right\} \quad j = d, d-1, \cdots, 0,$$

the computational part of which can be naïvely implemented as in Fig. 7.

```
      DO 1, J = D-1, 0, -1
        DO 2, I = D-J-1, -1
2         B(I,J) = A(I,J) + X*B(I+1,J)
1       B(0,J) = B(0,J) + Y*B(0,J+1)
```

FIG. 7. *The naïve, bivariate Horner code.*

The data dependency graph for this nested loop has vertices in bijective correspondence with the lower, triangular region $\{(i, j) | 0 \leqq i \leqq j \leqq d\}$, having flow dependencies from left to right between every pair of adjacent, horizontal vertices, and from top to bottom between every pair of adjacent, vertical vertices lying on the 0th vertical column. A corresponding, parallelising procedure similar to that of the full, univariate Horner can now be applied again. This time, one readily recognizes that the lines $i+j = \text{const.}$ are devoid of data dependencies, and provide optimal parallelisation in that no other choice of lines has fewer parallel translates with nonvoid intersections with the vertex set. A possible, parallelising transformation therefore corresponds to the unimodular change of variables $k = i+j$, and $l = j$. The transformed code now takes the form shown in Fig. 8.

```
      DO 1, K = D, 0, -1
        DO 2, L = 0, K
2         B(K-L,L) = A(K-L,L) + X*B(K-L+1,L)
1       B(0,K) = B(0,K) + Y*B(0,K+1)
```

FIG. 8. *The revised, bivariate Horner code.*

Note that each iteration of K requires three successive steps: the simultaneous multiplications, $x \cdot b_{k-l+1,l}$, for each value of l, concurrently with the multiplication of y with b_{0l+1}; the simultaneous additions of the products to the $a_{k-l,l}$; and finally, the addition on the last line. In general, where there are n indeterminates, all the multiplications can still be performed simultaneously, while the n additions can be performed in logarithmic time. The resultant complexity of the parallelised algorithm is therefore $(d+1)(\lceil \log_2 n \rceil + 1])$, using $d+1$ processors.

The revised Horner code admits another interpretation as a linear recurrence formula as follows. Define $e^{(k)}$ and $a^{(k)}$ to be $(d+1)$-dimensional vectors by setting

$$e_l^{(k)} = \begin{cases} \text{values B(L,K-L)} & \text{if } l < k. \\ 0 & \text{otherwise,} \end{cases} \quad \text{and}$$

$$a_l^{(k)} = \begin{cases} a_{l,k-l} & \text{if } l < k, \\ 0 & \text{otherwise.} \end{cases}$$

Then, for $k = d, d-1, \cdots, 0$,

$$e_l^{(k)} = xe_{l+1}^{(k+1)} + a_l^{(k)} \quad \text{for } l = 0, 1, \cdots, k-1, \quad \text{and}$$
$$e_k^{(k)} = ye_{k+1}^{(k+1)} + xe_k^{(k+1)} + a_k^{(k)},$$

which, in turn, can be represented as a block, tridiagonal system of linear equations, with the $(d+1)\times(d+1)$ identity matrices I appearing along the main diagonal, the diagonal matrices $-xI$ along the first subdiagonal, and with $-yE_{dd}$ along the second subdiagonal. Here, E_{dd} denotes the matrix whose only nonzero element is a one in the bottom right-hand corner.

Applying cyclic reduction blockwise, it is not difficult to see that each reduction gives rise to a new system having half the number of blocks, but with $-x^kI$ and $-y^kE_{dd}$ accumulating along the first and second subdiagonals, respectively. The main difference between the bivariate and univariate cases is that, for the bivariate case, the components of the $e_k^{(k)}$ are coupled with *two* others, so that an extra addition is required for the back substitutions. More generally, evaluating a degree d polynomial in n indeterminates entails an n fold coupling, and hence an additional $\lceil \log_2 n \rceil$ parallel steps during the back substitution phase.

For polynomials with n indeterminates, the iterations corresponding to values of the index variables lying on the parallel hyperplanes $H_k = \{(i_1, \cdots, i_q) | \sum_{q=1}^{n} i_q = k\}$ depend only upon those of the previous hyperplanes H_{k+1}, so that cyclic reduction applies and reaches completion after $2\lceil \log_2 (d+1) \rceil$ iterations. Each iteration requires a single, parallel multiplication step, and n additions. The resulting complexity of the logarithmically reduced algorithm is therefore $2\lceil \log_2 (d+1) \rceil \cdot (\lceil \log_2 n \rceil + 1)$; at most $O(d^n)$ processors are required.

5. The full, multivariate, Horner algorithm. Computing all the derivatives of the bivariate polynomial p of the last section corresponds to implementing the following, linear recurrence formulae:

$$b_{jk}^{(-1)} = a_{jk} \quad \text{for all } 0 \leqq j \leqq k \leqq d,$$
$$b_{jk}^{(i)} = 0 \qquad \text{for } 0 \leqq j \leqq k \leqq d+1,\ k = 0, \cdots, d \text{ and for } i = -1,\ j+k = d+1,$$

and

$$\left.\begin{aligned} b_{jk}^{(i)} &= b_{jk}^{(i-1)} + x \cdot b_{j+1k}^{(i)}, \qquad 0 \leqq k \leqq d-i, \quad i < j \leqq d-k, \\ b_{ik}^{(i)} &= b_{ik}^{(i-1)} + x \cdot b_{ii+1}^{(i)} + y \cdot b_{ik+1}^{(i)}, \qquad 0 \leqq k \leqq d-i, \\ b_{jk}^{(i)} &= b_{jk}^{(i-1)} + y \cdot b_{jk+1}^{(i)}, \qquad 0 \leqq j \leqq i-1, \quad d-j \leqq k \leqq i-j \end{aligned}\right\} 0 \leqq i \leqq d.$$

It is not difficult to show that $i!j!b_{ij}^{(i+j)} = \partial^{i+j}p/\partial x^i \partial y^j$. The significant section of such recurrence formulae can be programmed as in Fig. 9.

```
      DO 1, I = 0, D
        DO 2, K = 0, D-I
          DO 2, J = D-K, I + 1, -1
2           B(I,J,K) = B(I-1,J,K) + X*B(I,J+1,K)
        DO 3, J = I-1, 0, -1
3         B(I,J,K) = B(I-1,J,K) + X*B(I,J+1,K) + Y*B(I,J,K+1)
        DO 1, J = 0, I-1
          DO 1, K = D-J, I-J, -1
1           B(I,J,K) = B(I-1,J,K) + Y*B(I,J,K+1)
```

FIG. 9. *The naïve, full, bivariate Horner code.*

The vertices of the corresponding data dependence graph correspond to the region in $\mathbf{N}^3$ bounded by the planes $i=0$, $j=0$, $k=0$, $j+k=d$, and $j+k-i=0$. There are flow dependencies between adjacent vertices from the back to the front, and from the top to the bottom, and also from the right to the left. The optimal, parallelising hyperplanes are the hyperplanes $j+k-i=r$, for some constant $0 \leqq r \leqq d$. In the general case, the corresponding hyperplanes are given by the equation $\sum_{q=1}^{n} i_q - i = \text{const}$. The hyperplane parallelisation procedure reduces the parallel complexity of the full, multivariate Horner algorithm to $d+1$ iterations, each of which requires $\lceil \log_2 n \rceil$ parallel additions (cf. Fig. 10 below).

```
      DO 1, R = D, 0, -1
        DO 1, S = R, D
          DO 1, T = 0, S
1           B(S-R,S-T,T) = B(S-R-1,S-T,T) + X*B(S-R,S-T+1,T) + Y*B(S-R,S-T,T,T+1)
```

FIG. 10. *The revised, full, bivariate Horner code.*

To apply cyclic reduction once again, it is first necessary to represent the values of B(R,S,T) appropriately as a vector, whereupon one observes that the resulting linear difference equation is a block banded system, with n subdiagonal blocks, each of which is a shift operator. Although the details are now unpleasant, it can now nevertheless be seen that $2\lceil \log_2 (d+1) \rceil (\lceil \log_2 (d+1) \rceil + \lceil \log_2 n \rceil + 1)$ parallel steps are required, while using $O(d^{n+1})$ processors.

6. Conclusion. The arguments presented here are evidence for the effectiveness of considering the data themselves as the measure of parallelisability of an algorithm, and the use of dependence graphs in algorithm analysis. The univariate and bivariate linearised algorithms were both implemented on the Cray-XMP in Berlin, where, as predicted, not only were the revised codes fully vectorised, but they also yielded the same results to the point of replicating the numerically insignificant digits of the naïve code.

The table below shows the timing results of the naïve and revised univariate Horner codes. The former predictably has a quadratic execution time, whereas the latter is almost linear. Any deviation from linearity is due to the fact that a vector computer is not a genuine, parallel computer, since it still executes its instructions strictly sequentially. Since the Cray compiler is not capable of vectorising more than just the innermost loop, it is not sensible to time algorithms whose codes have a higher nesting order, with two or more inner loops parallelised. For this reason, the full logarithmic complexity of the multivariate evaluation algorithm could not be tested. The times below are given in microseconds.

Degree:	16	32	48	64	80	96	112	128	144	160	178	192
Naïve:	67	218	454	776	1183	1676	2253	2916	3665	4500	5419	6432
Revised:	24	49	78	110	149	187	229	273	323	376	431	488

REFERENCES

[1] U. BANERJEE, *Speed-up of ordinary programs*, Ph.D. thesis, University of Illinois, 1979.

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

[3] M. L. DOWLING, *A mathematical theory for code parallelisation*, Ph.D. thesis, Carolo-Wilhelmina Universität zu Braunschweig, Braunschung, FRG, 1987.
[4] P. HENRICI, *Applied and Computational Complex Analysis, Vol.* 1, John Wiley, New York, 1974.
[5] W. G. HORNER, *Philosophical Transactions of the London Mathematical Society*, 109 (1819), pp. 308–335.
[6] L. HYAFIL, *On the parallel evaluation of multivariate polynomials*, SIAM J. Comput., 8 (1979), pp. 120–123.
[7] D. E. KNUTH, *The Art of Computer Programming, Vol.* 2: *Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.
[8] D. KERSHAW, *Solution of single, tridiagonal systems and vectorisation of the* ICCG-*Algorithm on the* Cray-1, in Parallel Computations, G. Rodrigue, ed., Academic Press, New York, 1982.
[9] L. KRONSJÖ, *Algorithms: Their Complexity and Efficiency*, 2nd ed., John Wiley, New York, 1987.
[10] D. J. KUCK, R. H. KUHN, B. LEASURE, AND M. WOLFE, *Advanced, retargetable vectoriser*, IEEE Tutorial for Super-Computers: Design and Applications, K. Hwang, ed., 1984, pp. 186–203.
[11] L. LAMPORT, *The parallel execution of* DO-*loops*, Comm. ACM, 17 (1974), pp. 83–93.
[12] W. RÖNSCH, *Stability aspects in using parallel algorithms*, Parallel Comput., 1 (1984), pp. 75–98.
[13] G. RODRIGUE, N. MADSEN, AND J. KARUSH, *Odd-even reduction for banded linear equations*, J. Assoc. Comput. Mach., 26 (1979), pp. 72–81.
[14] L. G. VALIANT, S. SKYUM, S. BERKOWITZ, AND C. RACKOFF, *Fast parallel computation of polynomials using few processors*, SIAM J. Comput., 12 (1983), pp. 641–644.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 143-155, February 1990

009

VERY SIMPLE METHODS FOR ALL PAIRS NETWORK FLOW ANALYSIS*

DAN GUSFIELD†

Abstract. A very simple algorithm for the classical problem of computing the maximum network flow value between every pair of nodes in an undirected, capacitated n node graph is presented; as in the well-known Gomory-Hu method, the method given here uses only $n-1$ maximum flow computations. Our algorithm is implemented by adding only five simple lines of code to any program that produces a minimum cut; a program to produce an *equivalent flow tree*, which is a compact representation of the flow values, is obtained by adding only three simple lines of code to any program producing a minimum cut. A very simple version of the Gomory-Hu *cut tree* method that finds one minimum cut for every pair of nodes is also derived, and it is shown that the seemingly fundamental operation of that method, node contraction, is not needed, nor must crossing cuts be avoided. As a result, this version of the Gomory-Hu method is implemented by adding less than ten simple lines of code to any program that produces a minimum cut. The algorithms in this paper demonstrate that a cut tree of graph G can be computed with $n-1$ calls to an oracle that alone knows G, and that, when given two nodes s and t, returns any arbitrary minimum (s, t) cut and its value.

Key words. network flow, combinatorial optimization

AMS(MOS) subject classifications. 90B10, 90B35, 90C35, 68Q25, 05C99

1. Introduction.

For an undirected graph G with n nodes, Gomory and Hu [GH] showed that the flow values between each of the $n(n-1)/2$ pairs of nodes can be computed by solving only $n-1$ network flow problems on G, saving a factor of n over the obvious method. Furthermore, they showed that the flow values can be represented by a weighted tree T on n nodes, where for any pair of nodes (x, y), if e is the minimum weight edge on the path from x to y in T, then the maximum flow value from x to y in G is exactly the weight of e. Such a tree is called an *equivalent flow* tree. They also showed a stronger result, that there exists an equivalent flow tree, where for every pair of nodes (x, y), if e is as above, then the two components of $T-e$ form a minimum cut between x and y in G. Such a tree is called a *GH cut tree*, and it compactly represents one minimum cut for each pair of nodes. Figure 1 shows a three node graph G, a cut tree T of G, and an equivalent flow tree T' of G. Note that T' is not a cut tree of G. The method given in [GH] produces a GH cut tree using only $n-1$ maximum flow computations. This method is well known and is discussed in many texts and surveys on graphs and network flows [H1], [H2], [LP], [FF], [FR, FR], [LP], [HA], [PG], [VL], as well as in technical papers which build on it [AMS], [AH], [E], [H3],

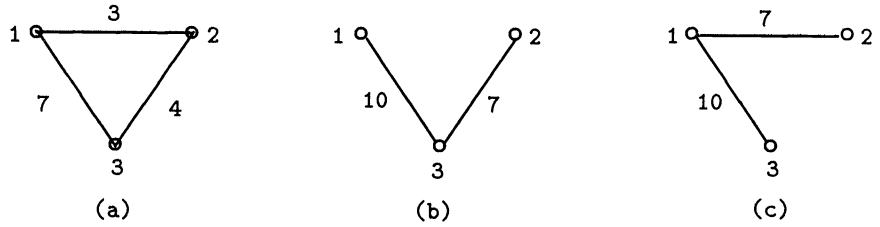


FIG. 1. *Graph G, a cut tree T, and an equivalent flow tree T'.*

* Received by the editors August 17, 1987; accepted for publication (in revised form) April 7, 1989. This research was partially supported by U.S. Census Bureau grant JSA 86-9 and National Science Foundation grant CCR-880374.

† Computer Science Division, University of California at Davis, Davis, California 95616.

[HR], [HS], [SC], [S], [T], [GrH]. For a basic discussion of graphs and network flows, see [FF], [L], or [H2]. For a textbook discussion of the GH method, see [H2] or [FF].

Two cuts (X, Y) and (U, V) are said to *cross* if all four set intersections, $X \cap U$, $X \cap V$, $Y \cap U$, and $Y \cap V$, are nonempty. The Gomory-Hu method, and methods based on it, require that all the cuts computed be pairwise noncrossing. Most of the work of the method, other than the work involved in the maximum flow computations, is involved in explicitly maintaining the noncrossing condition, or is a consequence of that condition. In particular, the operations of node contraction and identification of which nodes to contract, are consequences of the need to maintain noncrossing cuts. In all discussions of the GH method that we know of, both algorithmic and mathematical, the existence of noncrossing cuts has been fundamental to both the logic of cut trees, and to the algorithms to find and use them.

The GH method is fairly involved and nontrivial to program. A different method for computing all the flow values, and a cut tree, can be obtained by modifying a method of Schnorr [SC] for a related problem on directed graphs. This method requires $O(n \log n)$ maximum flow computations, but it can be implemented to have an amortized total running time of $O(n^4)$. However, the implementation is more complex than the GH method, and to obtain the faster time bound, or to build cut trees, the method also needs to maintain noncrossing (directed) cuts.

As for equivalent flow trees, in most of the published literature a full GH cut tree is used even when only the flow values are required. However, after the results in this paper were first obtained [GU1], we learned of a related method by Granot and Hassin [GrH] which can easily be modified to produce an equivalent flow tree, but not a cut tree. That method solves only $n-1$ maximum flow problems, and does not need to maintain noncrossing cuts. Hence, that is the first paper we know of that indicated that crossing cuts can be used in computing equivalent flow trees.

In this paper we give simple, efficient methods which show that crossing cuts can be used in producing GH cut trees as well as equivalent flow trees. We first give an extremely simple, efficient algorithm for producing an equivalent flow tree that is not necessarily a cut tree; as in the GH method, only $n-1$ maximum flows are computed by the method. The simplicity of the method comes from the fact that the method does not need to avoid crossing cuts, and so does not need to contract nodes. We implement the method by adding only three simple lines of code to any maximum flow program that produces a minimum cut; the program can be extended to explicitly output the $n(n-1)/2$ flow values, by adding only two additional lines of code. We next show that with a modification of the Gomory-Hu cut tree method, noncrossing cuts need not be maintained, and so the fundamental operation of node contraction is not needed, and the intermediate cut trees need not be explicitly represented or searched. Hence, the major programming and data structures details needed for the original GH method can be avoided. As a result, any maximum flow program producing a minimum cut can be converted to one that efficiently computes a GH cut tree, with the addition of under ten simple lines of code. More generally, we show that noncrossing cuts, which are central to all previous expositions on cut trees, are never explicitly needed in efficient algorithms for finding either cut trees or equivalent flow trees.

2. Equivalent flow trees and all pairs maximum flow.

ALGORITHM EQ. Input to the algorithm is an undirected capacitated graph G; output is an equivalent flow tree T'. The algorithm assumes the ability to find a minimum cut between two specified nodes in G.

1. Create a (star) tree T' on n nodes, with node 1 at the center and nodes 2 through n at the leaves.
2. For s from 2 to n do steps 3 and 4.
3. Compute a minimum cut (X, Y) in G between (leaf) node s and its (unique) neighbor t in T'. Label the edge (s, t) in T' with the capacity of (X, Y).
4. For every node i *larger than* s, if i is a neighbor of t, and i is on the s side of (X, Y), then modify T' by disconnecting i from t, and connecting i to s. Note that each node i larger than s remains a leaf in T'.

It is easy to see that at every iteration, node s and all nodes larger than s are leaves in T', so each chosen s has a unique neighbor, as expected by the algorithm. Figure 2 gives an example of the algorithm. Figure 2(a) shows the graph G, and the five cuts used by the algorithm; the capacity on each edge in G is one. Figure 2(b) shows tree T' before any cuts are computed; Figure 2(c) shows the tree after the first cut (1, 2) is computed; Figure 2(d) shows the final equivalent flow tree for G. Note that in this example the (5, 1) and the (3, 1) cuts each cross the (1, 2) cut. Also note that the equivalent flow tree T' of Fig. 1 would be obtained from running Algorithm EQ on the graph G of Fig. 1, illustrating the fact that Algorithm EQ does *not* always produce a cut tree.

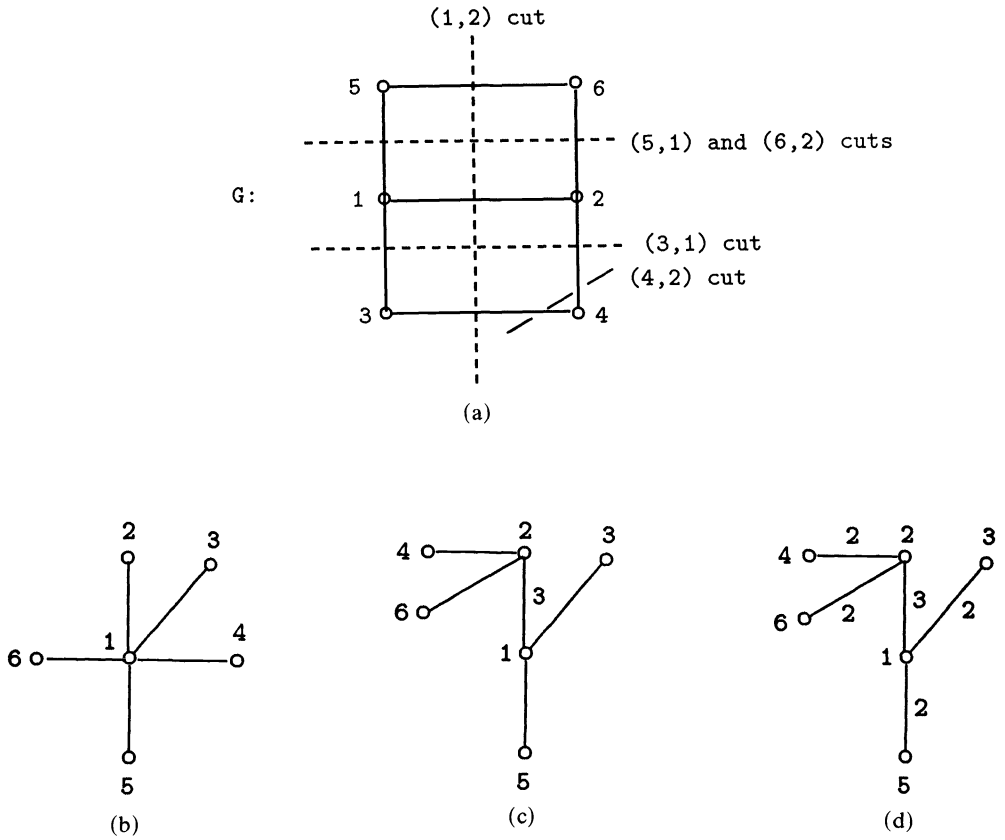


FIG. 2. *Graph G, and the creation of equivalent flow tree T' for G.*

To show the extreme simplicity of this method, we present the following "program" which implements Algorithm EQ. In the program, p is an n length vector initialized to 1; at every iteration, every node i larger than or equal to s is a leaf, and $p[i]$ indicates its unique neighbor. The program takes in graph G and outputs a set of weighted edges which form an equivalent flow tree T' of G.

PROGRAM EQ.

```
for s:=2 to n do
 begin
  Compute a minimum cut between nodes s and t:=p[s] in G;
  let X be the set of nodes on the s side of the cut.
  Output the edge (s, t) and the maximum s, t flow value f(s, t).
 for i:= s to n do
  if (i is in X and p[i]=t) then p[i]:=s;
 end;
```

To produce all the $n(n-1)/2$ flow values, let F be an n-by-n array, initialized to infinity, holding the flow values. Then insert the following lines before the "end;" above.

```
F[s, t]:=F[t, s]:=f(s, t);
for i:=1 to s-1 do
 if (i ⟨⟩ t) then F[s, i]:=F[i, s]:=min(f(s, t), F[t, i]);
```

In addition to the simplicity of the algorithm, it is noteworthy that the only interaction with graph G occurs inside the minimum cut routine. Hence, the algorithm can be thought of as $n-1$ calls to an oracle which alone knows the structure of G. Furthermore, for any given pair (s, t), if there is more than one minimum s-t cut, then the oracle (or adversary) is free to choose one arbitrarily. Thus, an equivalent flow tree for an unknown graph can be inferred from $n-1$ cut queries. We shall see that this is true for the cut tree as well.

We will present below a short, direct proof of the correctness of Algorithm EQ. A different, indirect, proof based on comparing the behavior of Algorithm EQ with the GH method is given in [GU1]. Before presenting the direct proof, we state some needed results initially shown in [GH].

LEMMA 1 [GH].[1] *Let (X, Y) be a minimum cut in G separating nodes $x \in X$ and $y \in Y$. Let u and v be two nodes on the X side of the cut, and let (U, V) be an arbitrary minimum (u, v) cut in G. If $y \in U$, then $(U', V') = (U \cup Y, V \cap X)$ is a minimum (u, v) cut, else (when $y \in V$) $(U', V') = (U \cap X, V \cup Y)$ is a minimum (u, v) cut.*

Figure 3 shows the two possibilities described by Lemma 1; cuts (X, Y) and (U, V) are drawn with straight solid lines, and cut (U', V') is drawn with a right angle, and marked by hatch marks. Note that in Lemma 1, it does not matter whether x is in U or in V; in Fig. 3 we have drawn x to be in U.

The importance of Lemma 1 is that it proves there always exists a minimum (u, v) cut (U', V') in G such that Y falls entirely on the u side or entirely on the v side of (U', V'). Hence (U', V') does not cross (X, Y). The existence of a noncrossing cut (U', V') is all that is needed in the correctness proof of the original GH method, but in this paper we use the following immediate, but key, corollary.

[1] The original lemma in [GH] is somewhat weaker, but the statement given here is explicitly stated and proved in the body of the proof of the original version. For the easiest such proof of Lemma 1, see [FF, p. 179] or [H2, pp. 66-68].

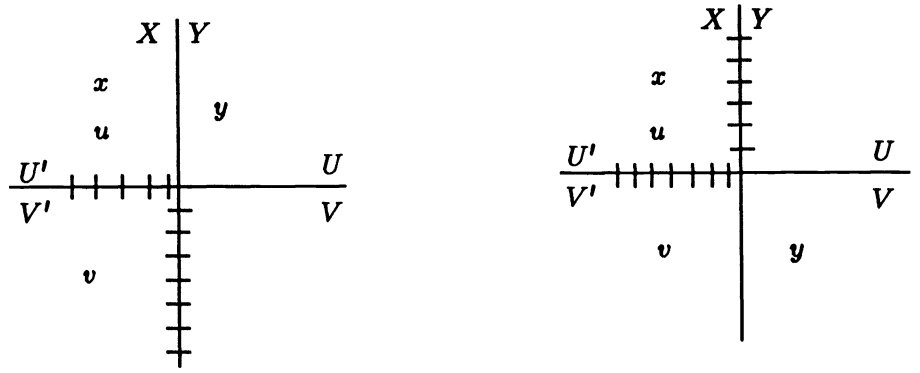


FIG. 3. *The two cases of Lemma* 1.

COROLLARY 1. *Let* (X, Y), (U, V), *and* (U', V') *be as in Lemma* 1. *Then the minimum* (u, v) *cut* (U', V') *does not cross* (X, Y), *and it splits* X *exactly the same way that* (U, V) *does.*

The following two facts are shown in [GH] (also in [FF] and [H2]) and are simple to prove.

LEMMA 2 [GH]. *Let* $f(x, y)$ *denote the maximum flow value between nodes* x *and* y. *If* $\{v_1, v_2, \cdots, v_k\}$ *is a set of nodes in* G, *then* $f(v_1, v_k) \geqq \min [f(v_i, v_{i+1})\colon i = 1 \text{ to } k-1]$.

COROLLARY 2 [GH]. *If* i, j, *and* k *are three arbitrary nodes in* G, *then the minimum of* $f(i, j)$, $f(i, k)$, *and* $f(j, k)$ *is not unique.*

2.1. Correctness of Algorithm EQ. Consider each edge (s, t) created in step 3 of the algorithm to be directed from s to t; then all edges are directed from larger node label to smaller node label, and hence T' is a directed tree where every directed path leads to node 1. For any path P (directed or not), let $\min(P)$ be the minimum weight of the edges on P.

LEMMA 3. *Suppose node* i *reaches node* j *by a directed path* $P[i, j]$ *in the final* T', *and suppose that* (k, j) *is a directed edge into* j, *where* k *is smaller (has smaller label) than any node on* $P[i, j]$ *except* j. *Then node* i *was a neighbor of* j *in* T' *at the time when the* (k, j) *cut* C *was computed by Algorithm* EQ. *Furthermore,* i *is on the* k *side of* C *if and only if* k *is on the directed path* $P[i, j]$ *in the final* T'.

Proof. At the start of the algorithm, node i is a neighbor of node 1 only. Then until iteration $i-1$, when i is node s in step 2 of the algorithm, node i has exactly one neighbor at any time, and the unique neighbor of i can change from v to w only when v is t and w is s in step 2. Hence every node on $P[i, 1]$ is a neighbor of i at some point before iteration $i-1$, and no node not on $P[i, 1]$ is. Then since $j < k$, j must be i's neighbor before the (j, k) cut C was computed. Furthermore, since k is smaller than every node on $P[i, j]$ except j, j must be the neighbor of i when C is computed. Now if k is on $P[i, j]$, then i surely is on the k side of C, and if k is not, then i cannot be on the k side. □

THEOREM 1. *Given input graph* G, *Algorithm* EQ *correctly computes an equivalent flow tree* T' *for* G.

Proof. First, note that if (x, y) is an edge in T', then Algorithm EQ computed an (x, y) minimum cut, and its value is written on edge (x, y). Hence the tree is correct for every pair of neighboring nodes in T'. Now we show that if (x, y) is an arbitrary pair of nodes not connected by an edge in T', and $P[x, y] = \{x = v_1, \cdots, v_k = y\}$ is the path (ignoring edge directions) in T' from x to y, then $f(x, y) = \min [f(v_i, v_{i+1})\colon i = 1 \text{ to } k-1]$. Given Lemma 2, we need only to show that $f(x, y) \leqq \min [f(v_i, v_{i+1})\colon i = 1$

to $k-1$]. Suppose not, and let (x, y) be the pair with shortest path $P[x, y]$ among all pairs where $f(x, y) > \min(P[x, y])$.

Case 1. Path $P[x, y]$ is a directed path from x to y (the case when it is directed from y to x is identical). Let $v \neq x$ be the neighbor of y on $P[x, y]$ (if $x = v$, the edge (x, y) is in T'). By the minimality of $P[x, y]$, $f(x, v) = \min(P[x, v])$, and since $f(x, y)$ is assumed to be greater than $\min(P[x, y])$, Corollary 2 implies that $\min(P[x, y]) = f(x, v) = f(v, y)$. But by Lemma 3, the cut between nodes y and v found by Algorithm EQ separates x and y, so $f(x, y) \leqq f(v, y) = \min(P[x, y])$, a contradiction.

Case 2. Path $P[x, y]$ consists of two directed subpaths $P[y, z]$ and $P[x, z]$, where $P[y, z]$ is directed from y to z and $P[x, z]$ is directed from x to z. Node z can be thought of as the least common ancestor of x and y in T' when node 1 is the root. Let x_1 be the neighbor of z on $P[x, z]$ and let y_1 be the neighbor of z on $P[y, z]$. Assume that $x_1 < y_1$, so in the running of Algorithm EQ the (x_1, z) cut, $C(x_1, z)$, was computed before the (y_1, z) cut.

From Case 1 we know that $f(x, z) = \min(P[x, z])$ and $f(y, z) = \min(P[y, z])$, so either $f(x, z)$ or $f(y, z)$ equals $\min(P[x, y])$. Hence by the assumption that $f(x, y) > \min(P[x, y])$, Corollary 2 says that $f(x, z) = f(y, z) = \min(P[x, y])$, and so there is an edge of weight $\min(P[x, y])$ on path $P[x, z]$. Let $e = (u, v)$ be the edge closest to z on $P[x, z]$ with weight $\min(P[x, y])$, let $C(u, v)$ be the (u, v) cut of that weight found by EQ, and let v be closer to z on $P[x, z]$ than u is. Then by Lemma 3, x, u, and v fall on the x_1 side of the cut $C(x_1, z)$ computed by the Algorithm EQ, and y falls on the z side of $C(x_1, z)$. By Lemma 3 again, x falls on the u side of $C(u, v)$, and from the assumption that $f(x, y) > \min(P[x, y])$, y must also fall on the u side. Figure 4 shows the general situation. In particular, the positions of nodes u, v, x, and y are each determined down to one of the four quadrants defined by the intersections of $C(x_1, z)$ and $C(u, v)$; the positions of nodes x_1 and z are each determined only to two quadrants.

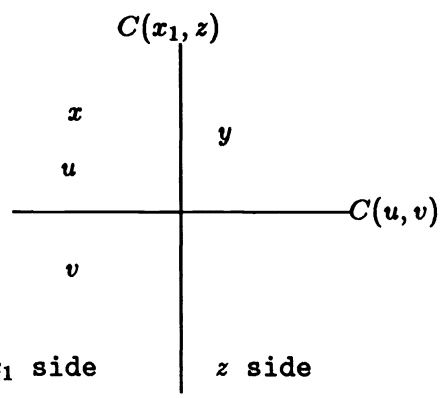


FIG. 4. *Case* 2 *of the proof of Theorem* 1.

Now there are two cases for the position of z. In either case, Lemma 1 can be applied (recall that in Lemma 1 the only assumption on the position of x_1 is that it is in X), yielding a minimum (u, v) cut C^* that either separates x and y, or that separates z and v. In particular, if $z \in U$, then the quadrant containing v defines a minimum (u, v) cut, and this cut also separates v and z; if $z \in V$, then the quadrant containing u defines a minimum (u, v) cut that also separates x from y. But, the minimum (u, v) cut has capacity $\min(P[x, y])$, so if C^* separates x and y, then $f(x, y) \leqq \min(P[x, y])$,

and so $f(x, y) = \min(P[x, y])$ as claimed. If C^* separates v and z, then $f(v, z) \leqq \min(P[x, y])$. But $P[v, z]$ is a directed path in T', so from Case 1, $f(v, z) = \min(P[v, z])$ and $\min(P[v, z]) > \min(P[x, y])$ by the selection of v, so $f(v, z) > \min(P[x, y])$. This gives a contradiction, and we conclude that $f(x, y) \leqq \min(P[x, y])$, so $f(x, y) = \min(P[x, y])$, and the correctness of Algorithm EQ is proved. □

3. A simple algorithm for the GH cut tree. In this section we show how to modify the GH method to avoid node contraction and the maintenance of noncrossing cuts. The result is a very simple algorithm to find a GH cut tree. The key idea is to show that although the original GH method must find in each step a minimum (u, v) cut that does not cross any previously used cuts, a modification of the method permits *any* minimum (u, v) cut to be used. The modified method will be proved correct by showing how its execution simulates a possible execution of the original GH algorithm.

DEFINITION. For a subset N_i of nodes of G, the *contraction* of N_i is the replacement of the nodes of N_i by a single node c_i, and for each node $v \in G - N_i$, the replacement of the edges from v to N_i with a single edge from v to c_i; the capacity of edge (v, c_i) is the sum of the capacities of the removed edges incident with v.

3.1. The Gomory–Hu method.

Input: An n node capacitated undirected graph G.
Output: A GH cut tree T for G.

1. Set T to be a single "supernode" containing every node of G. Then iterate the following step until every supernode contains only one node of G.
2. Pick a supernode S containing more than one node of G, and pick two nodes u and v in S. Find all the connected components of $T - S$ and let N_i be the set of nodes of G contained in the supernodes of the ith connected component of $T - S$. Successively contract the nodes in each set N_i in G, and let $G(S)$ be the resulting graph; note that the nodes in S are not contracted. Compute the maximum flow from u to v in $G(S)$. Let $f(u, v)$ be the value of the (u, v) flow, and let $C(u, v)$ be a minumum cut between u and v in $G(S)$. Let S_u be the supernode containing the nodes of G in S which fall on the u side of $C(u, v)$, and let S_v be the supernode containing the remaining nodes of S. Modify T by replacing supernode S with S_u and S_v, connected by an edge of weight $f(u, v)$. Any edge (S, S') incident with S in T is now moved to be incident with S_u if S' is in a contracted node of $G(S)$ on the u side of $C(u, v)$, and is moved to be incident with S_v if S' is in a contracted node of $G(S)$ on the v side of $C(u, v)$; note that the weights of all the edges remain unchanged, including those edges which were moved.

The existence of noncrossing cuts, stated earlier in Lemma 1, provides justification for the contraction operation in the GH method. That is, in order to find a minimum (u, v) cut in G, it is permissible to contract Y; a minimum (u, v) cut in the graph with Y contracted defines a minimum (u, v) cut in G, and of course, the two cuts have the same capacity. Applied iteratively from the leaves of T to S, the lemma can be used to show that a minimum (u, v) cut (for u and v in S) in the contracted graph $G(S)$, has the same *capacity* as a minimum (u, v) cut in G. Such a cut will of course not cross any previously found cuts, and is desired in the GH method because it is then easy to see how to use that cut to split S and how to reconnect the supernode neighbors of S to S_u and S_v.

3.2. Crossing cuts can be used to split a supernode. Consider the basic step in the GH method of dividing a supernode S by computing a minimum cut $C(u, v)$ between

u and v in the contracted graph $G(S)$. This step does two things: it decides how to split S into two new supernodes S_u and S_v, and it decides how to reconnect the neighbors of S to the supernodes S_u and S_v. In this section we will show how the GH method can use crossing cuts in carrying out the first decision.

DEFINITION. A pair of nodes (x, y) is called a *cut pair* for an edge e of an intermediate cut tree T if the nodes of G in the two connected components of $T-e$ form a minimum (x, y) cut in G.

For the following lemma, let T be an intermediate tree produced by the GH algorithm, with e an edge in T between two supernodes S and S'. Let (x, y) be a cut pair for edge e, with $x \in S$ and $y \in S'$; let u and v be any nodes in S, and let $C(u, v)$ be a minimum (u, v) cut in the contracted graph $G(S)$ defined from $T-S$. Let S_u and S_v be the new supernodes created from S, and let $\bar{T}$ be the updated intermediate tree given by the GH algorithm.

LEMMA 4 [GH].[2] *The pair (u, v) is a cut pair for the edge between S_u and S_v, in $\bar{T}$. Assume $x \in U$ (the case when $x \in V$ is symmetric). If (S', S_u) is an edge in $\bar{T}$, then (x, y) is a cut pair for it, and if (S', S_v) is an edge in $\bar{T}$, then (v, y) is a cut pair for it, in $\bar{T}$.*

Initially we will need only the following simpler version of Lemma 4, which follows easily by induction on the number of iterations of the GH algorithm.

COROLLARY 3 [GH]. *Let T be an intermediate tree in the computation of a* GH *cut tree, and let e be an edge in T between two supernodes S and S'. Then there is a pair of nodes (x, y) with $x \in S$ and $y \in S'$ such that (x, y) is a cut pair for e.*

Lemma 4 and its corollary are not as simple as they might at first seem, since x and y may not be the nodes used in the flow that created e, and the nodes that were used might not be in the current supernodes S or S' in $\bar{T}$.

We are now ready for the major theorem of this section.

THEOREM 2. *Let u and v be two nodes of G in supernode S of an intermediate* GH *tree T. If (U, V) is any minimum (u, v) cut in G (with $u \in U$ and $v \in V$), then there exists a minimum (u, v) cut (C_u, C_v) in the contracted graph $G(S)$ (with $u \in C_u$ and $v \in C_v$) such that $S \cap U = S \cap C_u$ and $S \cap V = S \cap C_v$, and such that the capacities of the two cuts are the same.*

Hence to determine how S could be split in a step of the GH method, we need not compute a cut in the contracted graph $G(S)$, but rather use the split of S created by a minimum cut splitting S in the original graph G.

Proof of Theorem 2. By Corollary 3, for each i from 1 to k, $C_i = (G - N_i, N_i)$ is a minimum cut separating some node in $G - N_i$ from some node in N_i, since $S \subseteq (G - N_i)$.

We now apply Corollary 1 to cuts C_1 and (U, V). Corollary 1 implies that there is a minimum (u, v) cut (U_1, V_1) with the same capacity as (U, V), such that $N_1 \subseteq U_1$ or $N_1 \subseteq V_1$, and such that $(G - N_1) \cap U = (G - N_1) \cap U_1$. Since $S \subseteq (G - N_1)$, it follows that $S \cap U = S \cap U_1$ (and $S \cap V = S \cap V_1$).

Now consider the cut $C_2 = (G - N_2, N_2)$. Since N_1 and N_2 are disjoint, and $S \subseteq G - N_2$, it follows that $S \cup N_1 \subseteq G - N_2$. Hence, by Corollary 1 there is a minimum (u, v) cut (U_2, V_2) derived from cuts C_2 and (U_1, V_1) such that

1. (U_2, V_2) has the same capacity as (U_1, V_1) and hence as (U, V).
2. $N_2 \subseteq U_2$ or $N_2 \subseteq V_2$.

[2] As with Lemma 1, the statement and proof of Lemma 4 is found in the body of a proof of a different proposition in [GH], [FF], and [H2]. The simplest such proof of Lemma 4 appears in [FF, p. 182] or [H2, pp. 71-73].

3. $(G-N_2)\cap U_2=(G-N_2)\cap U_1$, so $N_1\subseteq U_2$ or $N_1\subseteq V_2$ and $N_1\subseteq U_2$ if and only if $N_1\subseteq U_1$.
4. $S\cap U_2=S\cap U_1=S\cap U$ (and $S\cap V_2=S\cap V$).

Continuing in this way, using the fact that N_i is disjoint from S and from each $N_j: j\leqq i-1$, we can inductively apply Lemma 1 to cuts C_i and (U_{i-1}, V_{i-1}) (the cut obtained in iteration $i-1$) to obtain a minimum (u, v) cut (U_i, V_i) with the properties that

1. (U_i, V_i) has the same capacity as (U, V).
2. $S\cap U_i=S\cap U$ (and $S\cap V_i=S\cap V$).
3. $(G-N_i)\cap U_i=(G-N_i)\cap U_{i-1}$, so for all $j\leqq i$, $N_j\subseteq U_i$ or $N_j\subseteq V_i$ and $N_j\subseteq U_i$ if and only if $N_j\subseteq U_j$.

We conclude then that $S\cap U_k=S\cap U$ (and $S\cap V_k=S\cap V$), and that for each $i\leqq k$, $N_i\subseteq U_k$ or $N_i\subseteq V_k$, and (U_k, V_k) has the same capacity as (U, V). Now since each N_i is strictly on one side or the other of (U_k, V_k), it clearly defines a (u, v) cut (C_u, C_v) in $G(S)$ of the same capacity, and the theorem is proved. □

COROLLARY 4. *For all j, $N_j\subseteq U_k$ if and only if $N_j\subseteq U_j$.*

This corollary, and the last part of line labeled 3 above are not needed in the proof of Theorem 2, but will be needed later.

3.3. Reconnection despite crossing cuts. Theorem 2 shows how to determine, using the original G instead of a contracted graph, a split of S that the GH algorithm could have found. However, a minimum (u, v) cut C in G might split a set N_i between the u and v sides of C (i.e., might cross a previous cut); the GH algorithm has no rules to deal with such cuts. In this section we will see how to use crossing cuts to reconnect the neighbors of S to S_u and S_v.

3.3.1. Modifying the GH cut tree method. We first modify the GH method so that in every intermediate tree, every supernode S contains exactly one node called the *representative* of S, denoted $r(S)$. We start by arbitrarily declaring some node to be the representative of the first supernode of the GH method (the set of all nodes of G). We then impose the rule that when any supernode S is to be split, the flow computed must be between $r(S)$ and some other node v of S. After S is split into two supernodes $S_{r(S)}$ and S_v, $r(S)$ is the representative of $S_{r(S)}$, and v becomes the representative of S_v. It is then easy to see inductively that each supernode has exactly one representative. With this modification, successive application of Lemma 4 yields Lemma 5.

LEMMA 5. *Let T be an intermediate cut tree with S and S' any two adjacent supernodes in T; let N_i be the connected component of $T-S$ containing S'. Then $(G-N_i, N_i)$ is a minimum cut in G separating $r(S)$ and $r(S')$. That is, $(r(S), r(S'))$ is a cut pair for the edge in T between S and S'.*

For the statement of the following theorem, let S and N_j for $j\leqq k$ be as in Theorem 2, and for $j\leqq k$, let $y_j\in N_j$, $x_j\in(G-N_j)$ be such that $(G-N_j, N_j)$ is a minimum (x_j, y_j) cut in G (by Corollary 3, such an (x_j, y_j) exists). Also, for u and v in S, let (U, V) be *any* minimum (u, v) cut in G, and let (U_k, V_k) be the minimum (u, v) cut obtained from (U, V) as in the proof of Theorem 2.

THEOREM 3. *For a fixed j, if $x_j=u$, then $N_j\subseteq U_k$ if and only if $y_j\in U$.*

Proof. Corollary 4 says that $N_j\subseteq U_k$ if and only if $N_j\subseteq U_j$. So all that must be proved is that $N_j\subseteq U_j$ if and only if $y_j\in U$, assuming that $u=x_j$. Now if $u=x_j$, then $x_j\in U_{j-1}$ (since $S\cap U=S\cap U_{j-1}$), so Lemma 1 says that $y_j\in U_j$ if and only if $y_j\in U_{j-1}$. But $y_j\in N_j\subseteq(G-N_{j-1})$, and $(G-N_{j-1})\cap U_{j-1}=(G-N_{j-1})\cap U_{j-2}$, so $y_j\in U_{j-1}$ if and only if $y_j\in U_{j-2}$. Now $y_j\in(G-N_p)$ for all $p<j$, so we can induct as above to get

$(G - N_p) \cap U_p = (G - N_p) \cap U_{p-1}$, so $y_j \in U_p$ if and only if $y_j \in U_{p-1}$ for all $p < j$. Hence, assuming that $x_j = u$, it follows that $y_j \in U_k$ if and only if $y_j \in U$, and otherwise, $y_j \in V_k$. □

Theorem 3 is the key to reconnecting neighbors of S after S is split by a crossing cut.

COROLLARY 5. *For S a supernode in an intermediate tree T produced by the modified* GH *method, and for $v \neq r(S)$, let (U, V) be any minimum $(r(S), v)$ cut in G. The following rule correctly decides whether a neighbor of S, S', in T should be connected to $S_{r(S)}$ or to S_v: If $r(S')$ is on the $r(S)$ side of (U, V), then connect S' to $S_{r(S)}$, else to S_v.*

Proof. By Lemma 5, when the modified GH method is used, $r(S)$ satisfies the conditions required of x_j, namely, that $r(S) \in G - N_j$ the cut $(N_j, G - N_j)$ is a minimum $(r(S), r(S_j))$ cut, where S_j is the supernode neighbor of S in N_j. Furthermore, in the modified GH method, $u = x_j = r(S)$ for every j. Hence Theorem 3 implies that there exists a minimum (u, v) cut (U_k, V_k) in $G(S)$ such that for every j, $N_j \subseteq U_k$ if and only if $r(S_j) \in U$. Such a cut (U_k, V_k) could have been computed by the GH algorithm, and so the corollary follows. □

3.3.2. The method in brief. Theorem 2 and Corollary 5 form the basis of our simple version of the GH method. Initially, node 1 is the representative of the supernode consisting of all the nodes. When splitting a supernode S, compute an *arbitrary* minimum cut in G between $r(S)$ and any other node v in S. The nodes of S which fall on the v side of the cut form a new supernode S_v with representative v, and the other nodes in S remain in $S_{r(S)}$ with representative $r(S)$; if S' is a supernode neighbor of S in T before the split, and $r(S')$ falls on the v side of the cut, then replace the (S, S') edge with edge (S_v, S').

3.4. A simple complete cut tree program. To demonstrate the simplicity of our version of the GH method, we give the following program to compute a GH cut tree of input graph G. Theorem 2 and Corollary 5 allow great flexibility in the order in which supernodes are split, but for simplicity, the program below chooses s nodes in order from 2 to n. As in program EQ, p is an n length vector initialized to 1. At iteration s, $p[s]$ is the representative of the supernode that s is in. The edges of T are the final pairs $(i, p[i])$ for i from 2 to n, and edge $(i, p[i])$ has value $fl(i)$. If each edge is considered a directed edge from i to $p[i]$, then T forms a directed tree where every node leads to node 1.

CUT TREE PROGRAM MGH.

```
for s:=2 to n do
 begin
  Compute a minimum cut between nodes
  s and t:=p[s] in G; let X be the set of nodes on the s side
  of the cut. Output the maximum s, t flow value f(s, t).
 fl[s]:=f(s, t);
 for i:=1 to n do
  if (i<>s and i is in X and p[i]=t) then p[i]:=s;
 if (p[t] is in X) then
  begin
  p[s]:=p[t];
  p[t]:=s;
  fl[s]:=fl[t];
  fl[t]:=f(s, t);
  end;
 end;
```

We use the convention that the name of a supernode is given by the name of its representative, and note that after iteration $i-1$, nodes 1 through i are representatives of supernodes, and no node $j>i$ is a representative node in supernode $p[j]$; so for every node $j>s$, $p[v]$ indicates the representative of the supernode that v is in. Every supernode other than 1 points (with the p vector) to exactly one other supernode, and hence if x is a supernode other than 1, then its neighbors consist of those supernodes pointing to x, plus $p[x]$, the supernode to which x points. The neighbors of supernode 1 are just those supernodes with p value 1, i.e., those supernodes that point to 1. During the ith iteration, node $i+1$ becomes the representative of a supernode labeled $i+1$, and all representatives which point to $p[i+1]$ and which fall on the $i+1$ side of the $(i+1, p[i+1])$ cut are now made to point to $i+1$. Since the intermediate trees are being kept in an n-length vector, not an adjacency list, the only subtle part of the program occurs after a flow from $s=i+1$ to $t=p[i+1]$ if t points to a supernode neighbor x of t, and x falls on the s side of the (s, t) cut. In that case we make t point to s, and s point to x; otherwise, s remains pointing to t.

To explicitly accumulate the maximum flow values between all the pairs, we simply add the same two lines of code shown after algorithm EQ; the lines are added just before the final *end.* This is correct, because the set of (s, t) flow pairs generated in MGH is clearly a set that could have been generated in EQ. This accumulation of flow values can also be shown to be correct strictly in the context of the GH method, but was not obvious and was observed only after the discovery of algorithm EQ. Without this observation, a simple $O(n^2)$ method to explicitly calculate the $n(n-1)/2$ flow values is to do depth first search on the final cut tree, so that when backing up from a node x to y, the flow $fl(y, z)$ from y to a descendent z of x can also be computed as the minimum of $fl(x, y)$ and $fl(x, z)$. While this depth first search is not difficult, it requires a change in how T is represented, and the above two-line approach is certainly much simpler.

Note that, as in Algorithm EQ, the only interaction with G is in the minimum cut routine, so the tree could be inferred from $n-1$ calls to an oracle which returns a minimum cut and its value.

Relation with Algorithm EQ. The modified GH method can be described in terms of Algorithm EQ. To compute the GH tree, change step 4 of Algorithm EQ to read:

4. For every node i *other than* s, if i is a neighbor of t, and i is on the s side of (X, Y), then modify T' by disconnecting i from t, and connecting i to s, *labeling the new* (i, s) *edge with the label from the old* (i, t) *edge.*

Phrases in italics show the differences between this step 4 and the step 4 of Algorithm EQ.

4. Additional comments and extensions. (1) It is easy to underestimate the amount of programming detail needed by the original GH method. In fact, the ideas leading to this paper partly began after a failed attempt to quickly implement the method. The implementation was made more difficult because we used existing code for finding the maximum flow, but we did not understand the code well, and we needed to modify it to implement graph contraction and expansion. With the modified GH method of this paper, we totally avoid these difficulties, since we never touch any of the existing code, and never touch the graph after it is input.

In addition to the obvious work involved in contraction, an implementation of the original GH method must do a fair amount of work implied by the need to do

contraction. It must maintain T in a way so that the connected components can be efficiently found, and so that the nodes of G contained in particular supernodes of T can be identified, both to split a supernode, and to properly contract the nodes of G contained in a component of $T-S$. It must also maintain information about the connected components of $T-S$, or it must reexpand components after a flow, so that it can determine which supernodes fall on the u side and which on the v side of the cut $C(u, v)$ in $G(S)$.

(2) The original GH method might run faster in practice than the modified method (although the worst case asymptotic time is the same), since the contracted graphs are smaller than the original graph. However, it is an empirical question whether the speedup in flow computation compensates for the work needed to implement contraction and all the associated work implied by contraction; contraction should be seen as a heuristic that might accelerate the performance of the program.

(3) Some of the ideas in this paper have been extended and used to study the structure of minimum cuts in three other settings. A GH cut tree represents at least *one* minimum cut for each pair of nodes in an undirected edge-weighted graph. In [GN1] we generalize the GH cut tree, showing how to efficiently and compactly represent *all* minimum cuts between each pair of nodes. Interestingly, our method is based on equivalent flow trees, rather than on cut trees, further extending the importance of efficient computation of equivalent flow trees. This work also connects to and builds on recent work by Matula [M] and by Mansour and Schieber [MS] on computing connectivity quickly. In related work [GN2] we show how to construct with $O(n)$ maximum flow computations a cut tree for weighted node cuts, rather than edge cuts. We also show how to compactly represent weighted edge cuts in a directed graph.

(4) Very recently, Cheng and Hu [CH] have further reduced the importance of noncrossing cuts in equivalent flow trees. In Algorithm EQ and in the algorithm from [GrH], crossing cuts are allowed, but the proofs of correctness still use the fact that noncrossing cuts exist. Cheng and Hu give a different method which uses only $n-1$ maximum flow computations, and can be used to produce equivalent flow trees, but not cut trees. However, its proof of correctness does not even depend on the existence of noncrossing cuts. Because of that, their method can be used to represent minimum cut values where the value of a cut is given by an arbitrary function, i.e., is not the sum of the edge capacities crossing the cut. It is not difficult then to use this method to improve the problem considered in Schnorr [SC]. For a pair of nodes (i, j) define $\beta(i, j)$ as the minimum of the flow in a directed graph from i to j, or from j to i. These β values are needed in several problems [GN2], [GU]. Schnorr shows, using a very clever idea, that all the pairwise β values can be computed with $O(n \log n)$ maximum flow computations on the original graph. He then modifies that method to show that, with contraction, those $O(n \log n)$ flows run in total time $O(n^4)$. However, using the method of [CH] with its relaxed notion of cut values, the β values can be computed using only $O(n)$ maximum flow computations [GN2]. Hence in Schnorr's problem, contraction can also be avoided without sacrificing efficiency.

5. Conclusion. We have shown how to efficiently construct equivalent flow trees and GH cut trees without finding or maintaining noncrossing cuts, hence without node contraction and its associated work. The main theoretical consequence is conceptual clarity: node contraction, which is presented in existing discussions of the GH method as the fundamental algorithmic idea, is in fact not fundamental to cut tree computation; it should be seen as a heuristic which might accelerate the running of the flow computations. Similarly, although the *existence* of noncrossing cuts remains central in

the *logic* of cut trees, they are not explicitly needed in the efficient *computation* of cut trees. An additional theoretical consequence is the fact that a cut tree can be inferred from $n-1$ queries of an oracle which alone knows the actual graph. On the practical side, the import of these observations is that they lead to very simple, efficient programs for computing equivalent flow trees and cut trees; most of the programming and data structure details of the original GH method become unnecessary when contraction is avoided.

REFERENCES

[AH] D. ADOLPHSON AND T. C. HU, *Optimal linear ordering*, SIAM J. Appl. Math., 25 (1973), pp. 403-423.

[AMS] S. AGARAWAL, A. K. MITTAL, AND P. SHARMA, *Constrained optimum communications trees and sensitivity analysis*, SIAM J. Comput., 13 (1984), pp. 315-328.

[CH] C. K. CHENG AND T. C. HU, *Maximum concurrent flow and minimum ratio cut*, Tech. Report CS88-141, University of California, San Diego, CA, December 1988.

[E] S. E. ELMAGHRABY, *Sensitivity analysis of multi-terminal network flows*, J. ORSA, 12 (1964), pp. 680-688.

[FF] L. R. FORD AND D. R. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[FR, FR] H. FRANK AND I. T. FRISCH, *Communication, Transmission and Transportation Networks*, Addison-Wesley, Reading, MA, 1972.

[GH] R. E. GOMORY AND T. C. HU, *Multi-terminal network flows*, SIAM J. Appl. Math., 9 (1961), pp. 551-570.

[GrH] F. GRANOT AND R. HASSIN, *Multi-terminal maximum flows in node capacitated networks*, Discrete Appl. Math., 13 (1986), pp. 157-163.

[GU] D. GUSFIELD, *A graph theoretic approach to statistical data security*, SIAM J. Comput. 17 (1988), pp. 552-571.

[GU1] ———, *Very simple algorithms and programs for all pairs network flow analysis*, Tech. Report cse-87-1, Division of Computer Science, University of California, Davis, CA, April 1987.

[GN1] D. GUSFIELD AND D. NAOR, *Extracting maximal information about sets of minimum cuts*, Tech. Report cse-88-14, Division of Computer Science, University of California, Davis, CA, September 1988.

[GN2] ———, *Generalized cut trees: Efficient algorithms and uses*, Tech. Report cse-89-5, Division of Computer Science, University of California, Davis, CA, March 1989.

[HA] W. HANSJOACHIN, *Ten Applications of Graph Theory*, D. Reidel, Boston, MA, 1984.

[H1] T. C. HU, *Integer Programming and Network Flows*, Addison-Wesley, Reading, MA, 1969.

[H2] ———, *Combinatorial Algorithms*, Addison-Wesley, Reading, MA, 1982.

[H3] ———, *Optimum communication spanning trees*, SIAM J. Comput., 3 (1974), pp. 188-195.

[HR] T. C. HU AND F. RUSKEY, *Circular cuts in a network*, Math. Oper. Res., 5 (1980), pp. 362-373.

[HS] T. C. HU AND M. T. SHING, *Multiterminal flows in outplanar networks*, J. Algorithms (1983), pp. 241-261.

[L] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.

[LP] L. LOVASZ AND M. D. PLUMMER, *Matching theory*, Ann. Discrete Math., 29 (1986), North-Holland, Amsterdam, the Netherlands.

[M] D. MATULA, *Determining edge connectivity in $O(nm)$*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, October, 1987.

[MS] Y. MANSOUR AND B. SCHIEBER, *Finding the edge connectivity of directed graphs*, J. Algorithms, 10 (1989), pp. 76-85.

[PG] D. PHILLIPS AND A. GARCIA-DIAZ, *Fundamentals of Network Analysis*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[SC] C. P. SCHNORR, *Bottlenecks and edge connectivity in unsymmetrical networks*, SIAM J. Comput., 8 (1979), pp. 265-274.

[S] Y. SHILOACH, *A multi-terminal minimum cut algorithm for planar graphs*, SIAM J. Comput., 9 (1980), pp. 219-224.

[T] L. E. TROTTER, JR. *On the generality of multi-terminal flow theory*, Ann. Discrete Math., 1 (1977), pp. 517-525.

[VL] J. VAN LEEUWEN, *Graph algorithms*, Tech. Report RUU-CS-86-17, Department of Computer Science, University of Utrecht, Utrecht, the Netherlands, October 1986.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 156–163, February 1990

010

ON THE EXPECTED CAPACITY OF BINOMIAL AND RANDOM CONCENTRATORS*

EDWARD R. SCHEINERMAN†

Abstract. Masson and Morris [G.M. Masson and S.B. Morris *IEEE Trans. Comput.* C-32 (1983), pp. 649–657] introduced the notion of *expected capacity* of a concentrator and explicitly computed the expected capacity of $\binom{n}{2}$-concentrators for values of n up to 15. In this article, tools from random graph theory are employed to find asymptotic expressions for the expected capacity for this class of concentrators. It is shown that the same results can be obtained by concentrators that are constructed at random. It also is shown that the expected capacity of random concentrators is slightly inferior to the expected capacity of Pippenger's modular concentrators [N. Pippenger, *Expected capacity of modular concentrators*, preprint], and that random concentrators have certain advantages over deterministic designs. Finally, it is shown that expected capacity of a concentrator is actually a useful performance measure because the capacity of almost all input sets is very near the expected capacity.

Key words. concentrator, random graph

AMS(MOS) subject classification. 05C80

1. Introduction.

A concentrator is an interconnection network with m inputs and n outputs. Internal switches in this network enable inputs to be connected to the outputs. The capacity of the concentrator is the greatest integer k such that every k-element subset of the m inputs can be connected along disjoint channels to k of the n outputs.

More formally, the concentrators we consider are bipartite graphs $\Gamma = (X \cup Y, E)$, where X represents the set of inputs, Y the set of outputs, and E the switches connecting inputs to outputs. Such concentrators are also known as single-stage sparse crossbar networks. The capacity is the largest k such that every k element subset of X can be matched into Y. This definition of capacity provides a worst-case bound on the number of inputs the concentrator can serve. In [5] a notion of expected capacity was introduced.

Let the graph $\Gamma = (X \cup Y, E)$ be fixed. For a given subset $K \subset X$ of the inputs, let $\mathrm{cap}_K(\Gamma)$ denote the maximum size of a matching from K into Y. The *expected capacity* $e_k(\Gamma)$, corresponding to input size k, is the average of $\mathrm{cap}_K(\Gamma)$ over all subsets $K \subset X$ of size k, each taken as equiprobable. That is,

$$e_k(\Gamma) = \frac{1}{\binom{m}{k}} \sum_{K \subset X : |K| = k} \mathrm{cap}_K(\Gamma).$$

In both [5] and [6], the authors consider rather sparse concentrators. In particular, each input is connected to exactly two outputs; thus the degree of each X-vertex is exactly 2. Therefore, given a (not necessarily bipartite) graph G, we define a concentrator (bipartite graph) $\Gamma = \Gamma(G)$ as follows: The inputs (X-vertices) of Γ correspond to the m edges of G and the outputs (Y-vertices) of Γ correspond to the

*Received by the editors October 17, 1988; accepted for publication (in revised form) February 17, 1989. This work was supported in part by the Office of Naval Research, contract N00014-85-K0622.

†Department of Mathematical Sciences, The Johns Hopkins University, 3400 N. Charles Street, Baltimore, Maryland 21218 (ers@crabcake.cs.jhu.edu).

n vertices of G. We have a switch connecting x to y (i.e., $xy \in E(\Gamma)$), provided the edge of G corresponding to x is incident with the vertex of G corresponding to y. See Fig. 1(a)–(c).

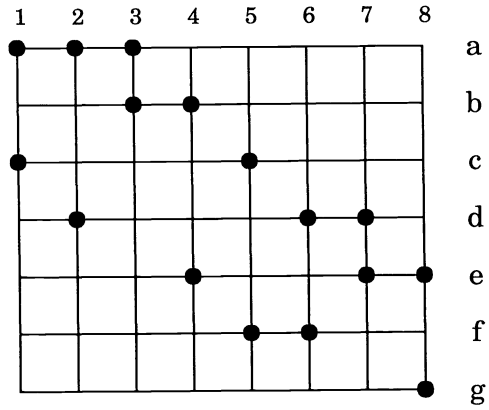


FIG. 1(a). *The graph G.*

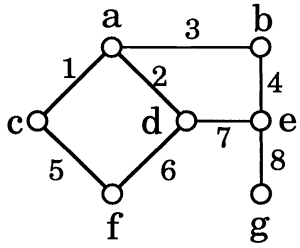


FIG. 1(b). *Concentrator* $\Gamma(G)$ *in bipartite graph form.*

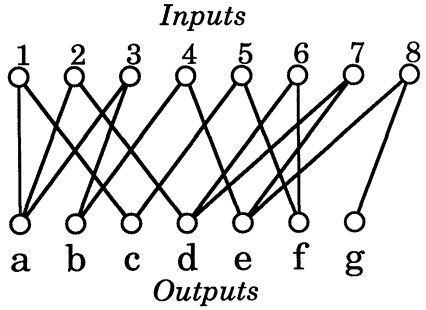


FIG. 1(c). *Concentrator* $\Gamma(G)$ *in crossbar form.*

In [5] and [6] the concentrators $\Gamma = \Gamma(G)$ arise from considering different choices for G. In [6] Pippenger chooses G as a particular 3-regular graph of high girth. These concentrators are called *modular* (3:2)-concentrators. They have $3n/2$ inputs and for

$k \leq 3n/4$, they have $e_k \sim k$. For larger values of k the expected capacity falls off a bit, but still gives good performance (see Table 1).

TABLE 1
Expected capacity of modular and random concentrators.

Input Size k	Expected Capacity e_k	
	Modular	Random
$0.10n$	$0.10000n$	$0.10000n$
⋮	⋮	⋮
$0.50n$	$0.50000n$	$0.50000n$
$0.60n$	$0.60000n$	$0.59630n$
$0.70n$	$0.70000n$	$0.67839n$
$0.75n$	$0.75000n$	$0.71335n$
$0.80n$	$0.79902n$	$0.74452n$
$0.90n$	$0.88148n$	$0.79686n$
$1.00n$	$0.93750n$	$0.83810n$
$1.10n$	$0.97115n$	$0.87060n$
$1.20n$	$0.98906n$	$0.89628n$
$1.30n$	$0.99709n$	$0.91665n$
$1.40n$	$0.99967n$	$0.93285n$
$1.50n$	$1.00000n$	$0.94579n$
$1.60n$	—	$0.95616n$
$1.70n$	—	$0.96448n$
$1.80n$	undefined	$0.97118n$
$1.90n$	for	$0.97659n$
$2.00n$	$k > 3n/2$	$0.98096n$
$2.50n$	—	$0.99314n$
$3.00n$	—	$0.99750n$

In [5] Masson and Morris use $G = K_n$, the complete graph on n vertices. These concentrators are called *binomial* or $\binom{n}{2}$-concentrators. Masson and Morris explicitly compute $e_k(\Gamma)$ for their concentrators with $0 \leq k \leq n \leq 15$. They found that e_k is generally very close to k.

In this paper we give asymptotic expressions for e_k in binomial concentrators. Further, we show that the expected capacity of binomial concentrators is the same as the expected capacity of "random" concentrators. We note that the expected performance of random concentrators is just slightly inferior to that of the modular (3:2)-concentrators of [6]. In the last section we show that expected capacity of a concentrator is actually a useful performance measure by showing that the capacity of almost all input sets is very near the expected capacity.

2. Expected capacity of $\binom{n}{2}$-concentrators. We first focus our attention on the crossbar considered in [5], $\Gamma = \Gamma(K_n)$ where n is large.

THEOREM 1. *Let c be a constant, $k = \lfloor cn/2 \rfloor$ and $\Gamma = \Gamma(K_n)$. If $c \leq 1$ then*

$e_k(\Gamma) \sim k$ *but if* $c > 1$ *then* $e_k(\Gamma) \sim [1 - u(c)]n$, *where*

$$u(c) = \frac{1}{c} \sum_{k=1}^{\infty} \frac{k^{k-2}}{k!} (ce^{-c})^k .$$

The key to proving this theorem is to apply known results from the theory of random graphs, a notion introduced by Erdős and Rényi in [3]. Given positive integers n and m, observe that there are $\binom{N}{m}$ labeled graphs on n vertices and m edges where $N = \binom{n}{2}$. Let $\mathcal{G}(n, m)$ denote the sample space of all such graphs on n vertices and m edges, each taken as equiprobable. A random graph is a graph drawn from the sample space $\mathcal{G}(n, m)$. Often we denote a randomly chosen graph in $\mathcal{G}(n, m)$ by G_m. For a property Q of graphs, we can ask for the probability that G_m satisfies Q. More often, we wish to know the limiting probability of $\Pr\{G_m \text{ has } Q\}$ as $n \to \infty$ and $m = m(n)$ depends on (and grows with) n. We say that almost every graph satisfies a property Q if this limit is 1.

The primary tools we need from random graph theory can be found in [2]. (See the following theorems.)

THEOREM 2. *Let* $0 < c < 1$ *be a constant and* $m = \lfloor cn/2 \rfloor$. *Almost every graph* G_m *is the union of tree and unicyclic components.*

THEOREM 3. *Let* $c > 1$ *be a constant,* $m = \lfloor cn/2 \rfloor$ *and* $\omega(n) \to \infty$. *Almost every* G_m *is the union of a "giant" component, tree components, and unicyclic components. The number of vertices in unicyclic components is at most* $\omega(n)$. *The number of vertices in the giant component is within* $\omega(n)\sqrt{n}$ *of* $[1 - t(c)]n$, *where*

$$t(c) = \frac{1}{c} \sum_{k=1}^{\infty} \frac{k^{k-1}}{k!} (ce^{-c})^k .$$

THEOREM 4. *Let* $c \neq 1$ *be a positive constant,* $m = \lfloor cn/2 \rfloor$, *and* $\omega(n) \to \infty$, *and let* $w = w(G_m)$ *denote the number of components in* G_m. *In almost every* G_m *we have that* w *is within* $\omega(n)\sqrt{n}$ *of* $u(c)n$, *where*

$$u(c) = \frac{1}{c} \sum_{k=1}^{\infty} \frac{k^{k-2}}{k!} (ce^{-c})^k .$$

The following lemma, which is from [6], enables us to compute $\text{cap}_K(\Gamma(G))$.

LEMMA 5. *Let* G *be a connected graph with* n *vertices and* m *edges, let* $\Gamma = \Gamma(G)$, *and let* K *denote the set of all inputs of* Γ. *If* G *is a tree, then* $\text{cap}_K(\Gamma) = m = n - 1$; *otherwise,* $\text{cap}_K(\Gamma) = n$.

Since this lemma is of central importance to our proof of Theorem 1 and is easy to prove, we paraphrase Pippenger's proof here.

Proof. We use induction on the number of vertices, n, in G. As the result is trivial in case $n = 0$, we assume the result has been proved for all graphs with fewer than n vertices. Let G be a graph with n vertices.

If G *has a vertex* v *of degree* 1, denote by e the edge incident with v. Let the input corresponding to e be matched to the output corresponding to v. We complete the matching by applying induction to $G - v$.

Otherwise, every vertex of G *has degree at least* 2. (Note that in this case G cannot be a tree.) In this case the X-vertices of $\Gamma(G)$ each have degree 2 and the Y-vertices of $\Gamma(G)$ each have degree at least 2. Thus for any subset $S \subset Y$ we have

that the neighbors of vertices in S, denoted $N(S) \subset X$ satisfies $|N(S)| \geq |S|$. (This holds because the number of edges emanating from S is at least $2|S|$ and the number of edges entering $|N(S)|$ from S is at most $2|N(S)|$.) Hence by Hall's marriage theorem, there is a matching in $\Gamma(G)$ that saturates the Y-vertices. □

If G is not connected, we perform the computation of Lemma 5 for each of its connected components.

Proof of Theorem 1. Let $\Gamma = \Gamma(K_n)$. Let K denote a set of k inputs, randomly chosen from the $\binom{N}{k}$ possible k-element subsets of the inputs (where $N = \binom{n}{2}$). Note that the inputs in K exactly correspond to a random selection of k edges of the K_n, giving us a random graph G_k. In applying the random graph theorems, we need to select a function ω of n as long as $\omega(n) \to \infty$. For example, take $\omega(n) = \log\log\log n$.

In case $c < 1$, then we know that the components of almost every G_k are trees and unicyclic graphs. By Lemma 5, it follows that $\mathrm{cap}_K(\Gamma) = k$ for such selections of K. Any other selection of K (in which the corresponding G_k has a different structure) appears with vanishing probability. Thus $e_k(\Gamma) = [1 - o(1)]k \sim k$.

In case $c > 1$, then almost every G_k has the following structure: one large component with $[1 - t(c)]n + O(\omega(n)\sqrt{n})$ vertices, $t(c)n + O(\omega(n)\sqrt{n})$ vertices in $u(c)n + O(\omega(n)\sqrt{n})$ tree components and at most $\omega(n)$ vertices in "small" unicyclic components. We now apply Lemma 5 to compute $\mathrm{cap}_K(\Gamma)$ when G_k has this structure. The inputs corresponding to edges in the "giant" component produce (asymptotically) $[1 - t(c)]n$ outputs, the inputs corresponding to the edges in the tree components produce (asymptotically) $[t(c) - u(c)]n$ outputs and the inputs corresponding to the edges in the unicyclic components produce a mere $\omega(n)$ outputs. Thus $\mathrm{cap}_K(\Gamma) = [1-u(c)]n+O(\omega(n)\sqrt{n})$. All other G_k's appear with vanishing probability and thus $e_k(\Gamma) \sim [1 - u(c)]n$.

Finally, we consider $c = 1$. Note that u is a decreasing continuous function of c and that $u(1) = 1/2$. Further, $e_k(\Gamma)$ is an increasing function of k. Thus for any $\epsilon > 0$ we can select a $\delta > 0$ so that if $k = \lfloor(1 - \delta)cn/2\rfloor$ then $e_k(\Gamma) \geq (1 - \epsilon)n/2$ and if $k = \lfloor(1 + \delta)cn/2\rfloor$ then $e_k(\Gamma) \leq (1 + \epsilon)n/2$. Thus when $c = 1$ we have $e_k(\Gamma) \sim \lfloor n/2\rfloor = k$. □

3. Random concentrators. We wish to construct a concentrator with m inputs, n outputs ($m > n$ and n large) in which each input is connected to exactly two outputs. In the previous section we considered the solution of [5] in which $m = \binom{n}{2}$. In [6] we have concentrators for $m = 3n/2$, where n is of the form $\binom{q+1}{3}$, where $q \geq 5$ is prime.

Here we consider concentrators designed at random. Specifically, given n and m our concentrator will be $\Gamma(G_m)$, where $G_m \in \mathcal{G}(n, m)$ is a random graph. Our first observation is that the average expected capacity of a random concentrator equals the expected capacity of $\Gamma(K_n)$.

PROPOSITION 6. *For $G_m \in \mathcal{G}(n, m)$ and $1 \leq k \leq m$ we have $E\{e_k[\Gamma(G_m)]\} = e_k[\Gamma(K_n)]$.*

Proof. It is convenient to work in a slightly altered probability space for our graphs. Let $\mathcal{G}' = \mathcal{G}'(n, m)$ denote the set of all graphs that have vertex set $V = \{1, \cdots, n\}$ and have m edges that are labeled (without repetition) by the integers $1, \cdots, m$. Since there are $m!$ ways to edge label the same graph, we have

$$|\mathcal{G}'| = |\mathcal{G}'(n, m)| = \binom{\binom{n}{2}}{m} m!$$

and each element of $\mathcal{G}'(n, m)$ is taken as equiprobable. Alternatively, we can think of a graph in $\mathcal{G}'(n, m)$ as a graph in $\mathcal{G}(n, m)$ together with an ordering of its edges. Notice

that whether we work in $\mathcal{G}'(n,m)$ or in $\mathcal{G}(n,m)$ we compute the same expectation for $e_k[\Gamma(G_m)]$. Let $K_0 = \{1, 2, \cdots, k\}$ denote a fixed set of inputs. Thus,

$$E\{e_k[\Gamma(G)]\} = \frac{1}{|\mathcal{G}'|} \sum_{G \in \mathcal{G}'} e_k[\Gamma(G)] \tag{1}$$

$$= \frac{1}{\binom{m}{k}|\mathcal{G}'|} \sum_{G \in \mathcal{G}'} \sum_{K} \mathrm{cap}_K[\Gamma(G)] \tag{2}$$

$$= \frac{1}{\binom{m}{k}|\mathcal{G}'|} \sum_{K} \sum_{G \in \mathcal{G}'} \mathrm{cap}_K[\Gamma(G)] \tag{3}$$

$$= \frac{1}{|\mathcal{G}'|} \sum_{G \in \mathcal{G}'} \mathrm{cap}_{K_0}[\Gamma(G)] \tag{4}$$

$$= E\{\mathrm{cap}_{K_0}[\Gamma(G)]\} \tag{5}$$

$$= e_k[\Gamma(K_n)]. \tag{6}$$

We have the equality (3) = (4) because the value of the sum

$$\sum_{G \in \mathcal{G}'} \mathrm{cap}_K[\Gamma(G)]$$

does not depend on K. The equality (5) = (6) holds because whether we randomly choose the inputs in $\Gamma(K_n)$ or fix the inputs and randomly choose the connections, we compute the same quantity. □

We wish to assert that almost all graphs $G \in \mathcal{G}(n,m)$ produce good concentrators $\Gamma(G)$. However, it is not enough to know that the mean expected capacity is good, since the good results may only be achieved on relatively few graphs. To show that almost every graph gives good concentrator behavior, we use the method of martingales.

Let $\{X_0, X_1, \cdots, X_m\}$ be a sequence of random variables defined on a common sample space. If $X_{i-1} = E[X_i | X_{i-1}]$ then the sequence is called a *martingale*. We use the following inequality due to Azuma [1] (see also [4]).

THEOREM 7. *Let $\{X_0, \cdots, X_m\}$ be a martingale and suppose that there exist constants $c_1, \cdots, c_m$ such that $|X_i - X_{i-1}| \leq c_i$, then*

$$\Pr\{|X_m - X_0| \geq \lambda\} \leq 2 \exp\left\{\frac{-\lambda^2}{2\sum_{i=1}^{m} c_i^2}\right\}.$$

We employ this inequality via the following definitions. For fixed k, define a random variable X on $\mathcal{G}'(n,m)$ by $X(G) = e_k[\Gamma(G)]$. Next, for $G \in \mathcal{G}'(n,m)$ denote by $\epsilon_j(G)$ the set of the *first* j edges in G (recall that the edge sets of graphs in $\mathcal{G}'(n,m)$ are ordered). Now we define our martingale $\{X_0, X_1, \cdots, X_m\}$ by

$$X_j(G) = E\left[X(H) | \epsilon_j(H) = \epsilon_j(G)\right].$$

In other words, $X_j(G)$ is the average of X over all graphs H whose first j edges agree with G. Thus X_0 has constant value $E(X)$ and $X_m = X$. Further, the martingale condition, $X_{i-1} = E[X_i | X_{i-1}]$, is satisfied. Finally, as we vary the connections of a single input, j, we can change the expected capacity by at most 1, hence we have $|X_j - X_{j-1}| \leq 1$. Applying Azuma's inequality we have

$$\Pr\{|X - E(X)| \geq \lambda\} \leq 2 \exp\left\{\frac{-\lambda^2}{2m}\right\}. \tag{7}$$

THEOREM 8. *Let $m = o(n^2/\log n)$. Almost every graph $G \in \mathcal{G}(n,m)$ is such that for all k with $1 \le k \le m$ we have*

$$e_k[\Gamma(G)] = \begin{cases} k + o(n) & \text{when } k \le n/2, \\ [1 - u(c)]n + o(n) & \text{when } k = cn/2 \text{ for constant } c > 1, \text{and} \\ n + o(n) & \text{when } k/n \to \infty \end{cases}$$

where, as before,

$$u(c) = \frac{1}{c} \sum_{k=1}^{\infty} \frac{k^{k-2}}{k!} (ce^{-c})^k .$$

Proof. Suppose $m = n^2/(\omega(n)\log n)$, where $\omega(n) \to \infty$. We know from Proposition 6 that for each k the mean expected capacity of $\Gamma(G)$ has the value asserted in the theorem. For fixed k, with $1 \le k \le m$, the probability that G does not have $e_k[\Gamma(G)]$ within $\lambda = 2n/\sqrt{\omega(n)} = o(n)$ of the mean can be estimated from inequality (7) to be

$$\Pr\{|E(e_k) - e_k| \ge \lambda\} \le 2\exp\left\{\frac{-2n^2/\omega(n)}{n^2/(\omega(n)\log n)}\right\} = O(n^{-2}).$$

The probability that G does not have $e_k[\Gamma(G)]$ within λ of the mean for *arbitrary* k can therefore be bounded by $mO(n^{-2}) \to 0$ as $n \to \infty$. □

We can compare expected capacity of a randomly generated concentrator with the concentrators proposed in [6]. These concentrators are referred to as modular (3:2)-concentrators $\Gamma(G)$ whose underlying graph G is 3-regular and of high girth. Thus the modular (3:2)-concentrators have $m = 3n/2$ inputs and n outputs. Pippenger [6] defines G using an algebraic construction. He shows that $e_k(\Gamma)$ for his concentrators is given by

$$e_k(\Gamma) = \begin{cases} k + o(n) & \text{when } k \le \frac{3}{4}n, \text{ and} \\ k - \frac{n}{2}\left(2 - \frac{3n}{2k}\right)^3 + o(n) & \text{when } \frac{3}{4}n \le k \le \frac{3}{2}n = m. \end{cases}$$

Thus the expected performance e_k of modular (3:2)-concentrators dominates that of random concentrators, but not by a wide margin. Table 1 shows the expected capacity e_k of modular and random concentrators for large n and various input sizes.

4. Justification of expected capacity. It is conceivable that the expected capacity of a concentrator can be high while the actual capacity of many inputs is low. Were this the case, the expected capacity might not be a useful performance measure for concentrators. Fortunately, we can apply the martingale method to show that the expected capacity is nearly achieved by almost all sets of inputs.

For fixed k, we consider all possible sets of k inputs to Γ as equiprobable. It will be convenient to view the input set K as an ordered k-tuple without repetitions, $K = (i_1, i_2, \cdots, i_k)$, each with probability $(m-k)!/m!$.

Let X denote the random variable defined on the sample space of all ordered inputs by $X(K) = \mathrm{cap}_K(\Gamma)$. Note that $E(X) = e_k(\Gamma)$. We now define a martingale by

$$X_j(K) = E\left[X(K') | i'_1 = i_1, i'_2 = i_2, \cdots, i'_j = i_j\right].$$

In other words, $X_j(K)$ is the average capacity of those ordered inputs that agree with K in the first j places. Note that X_0 has constant value $e_k(\Gamma)$ and that $X_k(K) =$

$X(K) = \text{cap}_K(\Gamma)$. Note that the sequence $\{X_0, \cdots, X_k\}$ is a martingale. Moreover, since changing a single input in K can change $\text{cap}_K(\Gamma)$ by at most 1, we have $|X_j - X_{j-1}| \leq 1$. Thus by Azuma's inequality (Theorem 7):

$$\Pr\{|\text{cap}_K(\Gamma) - e_k(\Gamma)| \geq \lambda\} = \Pr\{|X_k - X_0| \geq \lambda\} \leq 2\exp\left\{\frac{-\lambda^2}{2k}\right\}$$

which is negligible if $\lambda \gg \sqrt{k}$.

Thus high expected capacity translates into excellent performance for nearly all input sets.

Acknowledgment. The author wishes to thank Professor Gerald Masson for introducing him to this subject and for many interesting discussions.

REFERENCES

[1] K. AZUMA, *Weighted sums of certain dependent random variables,* Tôhoku Math. J., 19 (1967), pp. 357–367.
[2] B. BOLLOBÁS, *Random Graphs*, Academic Press, New York, 1985.
[3] P. ERDŐS AND A. RÉNYI, *On random graphs I,* Publ. Math. Debrecen 6 (1959), pp. 290–297.
[4] D. A. FREEDMAN, *On tail probabilities for martingales,* The Annals of Probability 3 (1975), pp. 100–118.
[5] G. M. MASSON AND S. B. MORRIS, *Expected capacity of* $\binom{m}{2}$*-networks,* IEEE Trans. on Comput. C-32 (1983), pp. 649–657.
[6] N. PIPPENGER, *Expected capacity of modular concentrators*, preprint.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 164–181, February 1990

011

SPACE-EFFICIENT MESSAGE ROUTING IN c-DECOMPOSABLE NETWORKS*

GREG N. FREDERICKSON† AND RAVI JANARDAN‡

Abstract. The problem of routing messages along near-shortest paths in a distributed network without using complete routing tables is considered. It is assumed that the nodes of the network can be assigned suitable short names at the time the network is established. Two space-efficient near-shortest path routing schemes are given for any class of networks whose members can be decomposed recursively by a separator of size at most a constant c, where $c \geq 2$. For an n-node network, the first scheme uses a total of $O(cn \log n)$ items of routing information, each $O(\log n)$ bits long, and $O(\log n)$-bit names, generated from a separator-based decomposition of the network, to achieve routings that are at most three times longer than shortest routings in worst case.[1] The second scheme augments the node names with $O(c \log c \log n)$ additional bits and uses this to reduce the bound on the routings to $(2/\alpha) + 1$, where α, $1 < \alpha \leq 2$, is the root of the equation $\alpha^{\lceil (c+1)/2 \rceil} - \alpha - 2 = 0$. For both schemes, the node names and the routing information can be determined efficiently.

Key words. distributed network, graph theory, k-outerplanar graph, routing, separator, series-parallel graph, shortest paths

AMS (MOS) subject classifications. 68M10, 68Q20, 68R10, 94C15

1. Introduction.

One of the primary functions in a distributed network is the routing of messages between pairs of nodes. Assuming that a nonnegative cost, or distance, is associated with each edge, it is desirable to route along shortest paths. While this can be accomplished using a complete routing table at each of the n nodes in the network, such tables are expensive for large networks, storing a total of $\Theta(n^2)$ items of routing information, where each item is a node name. Thus, recent research has focused on identifying classes of network topologies for which the shortest paths information at each node can be stored succinctly. It is assumed that the nodes can be assigned suitable short names at the time the network is established. The idea behind naming nodes is to encode useful information about the network in the node names and use this to do the routing. Shortest path routing schemes that use $O(\log n)$-bit node names and a total of $\Theta(n)$ items of routing information have been given for networks such as trees, unit-cost rings [SK],[vLT1], unit-cost complete networks, unit-cost grids [vLT2], and networks at the lower end of a hierarchy identified in [FJ1] (the simplest of which are the outerplanar networks [H]). Unfortunately, the approach in the above research becomes expensive even for very simply defined classes of networks such as, for instance, the series-parallel networks [D]. However, by shifting our focus to consider schemes that route along near-shortest paths, we have been able to design space-efficient routing schemes for much broader classes of network topologies.

The issue of saving space in routing tables by settling for near-shortest path

* Received by the editors December 11, 1986; accepted for publication (in revised form) April 14, 1989. A preliminary version of the results in this paper appeared as a part of *Separator-based strategies for efficient message routing*, Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, October 1986, pp. 428–437.

† Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by National Science Foundation grant CCR–86202271 and by Office of Naval Research contract N 00014-86-K-0689.

‡ Department of Computer Science, University of Minnesota, Minneapolis, Minnesota 55455. The research of this author was supported by National Science Foundation grants DCR–8320124 and CCR–8808574, and by a grant-in-aid of research from the Graduate School of the University of Minnesota.

[1] Unless stated otherwise, all logarithms are to the base 2.

routings was first raised in [KK]. (Indeed, this is the first reported work on the problem of space-efficient routing.) Networks of general topology were studied in [KK] and a clustering approach was proposed for naming the nodes. Unfortunately, no indication was given of how to do the clustering. Further, the routings produced depended crucially on certain strong assumptions about the structure of the clusters, and, in worst case, could be $O(n)$ times longer than shortest routings. In this paper, and in a related paper [FJ2], we consider various classes of networks that exhibit a certain separator property and we show how to take advantage of this property to design space-efficient near-shortest routing schemes. All our schemes achieve routings that are, in worst case, at most a small constant times longer than corresponding shortest routings. More recently, general networks with unit cost edges have been considered in [PU] and a trade-off has been established between the space used and the quality of the routings generated. Both upper and lower bounds are given for this trade-off.

In this paper, we present two near-shortest path routing schemes for any class of c-decomposable networks, defined as follows. Let the network be represented by an n-node undirected graph $G = (V, E)$. Consider an assignment of nonnegative weights to the nodes of G and let $c \geq 2$ be a constant. A *c-separator* of G for this weight assignment is a set C of at most c *separator nodes* whose removal partitions the remaining nodes into sets A and B, each containing at most two-thirds of the total weight and with no node in A adjacent to a node in B. We call G a *c-decomposable graph* if it has a c-separator for every assignment of weights to its nodes. Examples of c-decomposable graphs are the series-parallel graphs [D], for which $c = 2$, and the k-outerplanar graphs [B] where $k > 1$ is a constant, for which $c = 2k$. (The 1-outerplanar networks, or, more simply, the outerplanar networks, are also c-decomposable, for $c = 2$. However, we ignore them in this paper because an optimal routing scheme for these has already been given in [FJ1].) As we shall see, the c-decomposability of the network allows us to recursively apply a c-separator algorithm to perform a hierarchical decomposition of the network and assign suitable names to the nodes.

We measure the quality of the routings achieved by our schemes on a network by the *performance bound*, defined as the maximum ratio $\hat{\rho}(u, v)/\rho(u, v)$ taken over all pairs of nodes u, v in the network, where $\rho(u, v)$ is the length of a shortest path from u to v and $\hat{\rho}(u, v)$ is the length of the routing from u to v. Our first scheme, called the *basic routing scheme*, uses $O(\log n)$-bit names and a total of $O(cn \log n)$ items of routing information (where each item is $O(\log n)$ bits long) to achieve a performance bound of 3. The second scheme, called the *enhanced routing scheme*, incorporates in the node names of the basic scheme $O(c \log c \log n)$ additional bits of information about relative distances and uses this to achieve a performance bound of $(2/\alpha) + 1$, where α, $1 < \alpha \leq 2$, is the root of the equation $\alpha^{\lceil (c+1)/2 \rceil} - \alpha - 2 = 0$. Thus, the performance bound is 2 for $c \leq 3$ and ranges up to strictly less than 3 as c increases.

Our results also hold for classes of c-decomposable networks for which c is not a constant, but instead depends on n. An example is the class of planar networks, for which c is $O(\sqrt{n})$ [LT]. However, we will not consider such networks in this paper for two reasons. First, the techniques of this paper are geared specifically towards c-decomposable networks with constant c. When c is not a constant, it may be possible to do better by employing techniques different from those used in this paper, as is the case for planar networks [FJ2]. Second, when c is not a constant, node names in the enhanced routing scheme are no longer $O(\log n)$ bits long. Although c is a constant throughout this paper, we will include it within $O(\cdot)$ bounds on space and time, in

order to make explicit the exact dependence of these resources on c.

In [FJ2] two routing schemes are given for planar networks. The first scheme uses $O(\log n)$-bit names and $O(n^{4/3})$ items of routing information, each $O(\log n)$ bits long, to achieve a performance bound of 3. For any constant ϵ, $0 < \epsilon < 1/3$, the second scheme can be set up to use $O(n^{1+\epsilon})$ items of routing information, each $O((1/\epsilon)\log n)$ bits long, and achieve a performance bound of 7, but at the expense of $O((1/\epsilon)\log n)$-bit names. These schemes are also separator-based, the first using the separator strategy of [LT] and the second the more structured cyclic separator of [M]. However, owing to the comparatively larger size of the separator for planar networks, the techniques used for decomposition, naming, and routing are quite different from those in the current paper.

The rest of this paper is organized as follows. In the next section we describe how the network is decomposed hierarchically and how the nodes are assigned names. The basic scheme is given in §3 and the enhanced scheme in §4. Section 5 discusses efficient separator strategies for two specific classes of c-decomposable graphs, namely, series-parallel graphs and k-outerplanar graphs. Section 6 discusses how to set up the routing schemes efficiently.

2. Hierarchical decomposition and naming. We show how to generate suitable names for the nodes of G in order to facilitate the routing. The separator property is used to decompose G hierarchically into levels and to assign names to the nodes based on their relative positions in the decomposition.

The graphs at various levels are generated inductively as follows. The graph at level 0 is $G_0 = G$. Define the *core* of G_0 to be G_0 itself, and call each node in the core a *core node* of G_0. For $i \geq 0$, let G_ω be a level i graph, where ω is a binary string. If G_ω has more than c core nodes, then a c-separator algorithm is applied to G_ω after assigning equal positive weights to its core nodes and zero weights to its remaining nodes. (As we show later, G_ω is c-decomposable, so that a c-separator exists for the chosen weight assignment.) Let $G'_{\omega 0}$ and $G'_{\omega 1}$ be the subgraphs of G_ω induced on the vertex sets $A \bigcup C$ and $B \bigcup C$, respectively, where A, B, and C are as in the c-separator definition given previously.

The separation of G_ω into $G'_{\omega 0}$ and $G'_{\omega 1}$ may not preserve distances between core nodes of G_ω that end up in $G'_{\omega 0}$ (respectively, $G'_{\omega 1}$), since some of the shortest paths between these nodes may use portions of $G'_{\omega 1}$ (respectively, $G'_{\omega 0}$). As we shall see, these distances need to be preserved in order to achieve the claimed performance bounds. This is accomplished by augmenting $G'_{\omega 0}$ (respectively, $G'_{\omega 1}$) with a suitable graph derived from $G'_{\omega 1}$ (respectively, $G'_{\omega 0}$), which represents the shortest paths lost due to the separation. The size of the augmenting graph is kept small ($O(c^4)$ nodes and edges) so that the routing schemes can be set up efficiently. We note that information about the augmentation is not needed once the routing scheme has been set up, since the routing itself takes place in the actual network G. We discuss the augmentation in more detail later.

The augmentation of $G'_{\omega 0}$ and $G'_{\omega 1}$ yields the level $i+1$ graphs $G_{\omega 0}$ and $G_{\omega 1}$, respectively. Define the *core* of $G_{\omega 0}$ (respectively, $G_{\omega 1}$) as the subgraph of G_ω induced on the core nodes of G_ω that are in A (respectively, B). Call each such node a *core node* of $G_{\omega 0}$ (respectively, $G_{\omega 1}$). Any other node of $G_{\omega 0}$ (respectively, $G_{\omega 1}$) is a *noncore node.*

Because of the way the nodes of G_ω are weighted, it follows from the definition of a c-separator that the number of core nodes in each of $G_{\omega 0}$ and $G_{\omega 1}$ is at most two-thirds the number of core nodes in G_ω. Since G_0 has n core nodes and since no

graph with c or fewer core nodes is decomposed further, it follows that the number of levels in the decomposition is $O(\log_{3/2}(n/c))$, which is $O(\log n)$.

The decomposition establishes certain natural relationships between the nodes, as follows. If level i graph G_ω is decomposed further, then each separator node of G_ω that is also a core node of G_ω is a *level i node* in the decomposition. Otherwise, G_ω has at most c core nodes, and each is a *level i node.* Any two level i nodes that belong to the core of the same level i graph G_ω are *siblings.* Suppose that G_ω is decomposed further, into $G_{\omega 0}$ and $G_{\omega 1}$. Let u be a separator node of G_ω. If v is any core node $G_{\omega 0}$ or $G_{\omega 1}$, then u is an *ancestor of v for level i.* We further distinguish between ancestors as follows. If u is also a core node of G_ω, then u is a *real ancestor of v for level i.* If u is a noncore node of G_ω, then it is a *pseudo-ancestor of v for level i.* Note that it is possible for a node to be a real ancestor of another node for some level j and a pseudo-ancestor of that node for some other level $j' > j$. Two nodes are *related* in the decomposition if they are siblings or if one is a real ancestor of the other for some level; otherwise, they are *unrelated.*

Each level i node belonging to the core of G_ω is given the name ω, along with an integer distinguisher of value at most c, to make names distinct. Clearly, any name is $O(\log n)$ bits long. This naming has the property that two nodes are related if and only if the distinguisher-free portions of their names are identical or if one is a proper prefix of the other. For unrelated nodes u and v, if l is the length of the longest common prefix of the distinguisher-free portion of their names, then u and v are in the core of the same level $l-1$ graph, but are in the cores of different level l graphs resulting from the decomposition of the level $l-1$ graph. Level l is called the *separating level for u and v.* As we shall see, the separating level plays a crucial role in the routing strategy.

We illustrate the decomposition and naming in Fig. 1. The given graph G_0 is 4-decomposable. For simplicity, let all edge costs be 1. The separator nodes of G_0, which become the level 0 nodes in the decomposition, are shown filled in. Only one of the two graphs resulting from the separation of G_0, namely, G'_{00}, is shown, together with the names assigned to the level 0 nodes. The symbol "#" is a delimiter and the integer following it is the distinguisher. Any two of the named nodes are siblings and each named node is a real ancestor for level 0 of an unnamed node. Graph G_{00} is also shown, with the portion introduced by the augmentation shown dashed. The nodes of G_{00} that are not filled in and that have solid edges incident with them are the core nodes of G_{00}; the remaining nodes of G_{00} are its noncore nodes. In order to illustrate pseudo-ancestors, suppose that the node in the augmenting graph that is adjacent to node 0#1 becomes a separator node of G_{00} at the next level in the decomposition. Let this node be u and let v be the degree 3 node adjacent to 0#1. Then u is a pseudo-ancestor of v for level 1.

We now discuss how the augmentation is performed. Let $G'_{\omega 0}$ and $G'_{\omega 1}$ be the graphs resulting from the separation of G_ω. $G'_{\omega 0}$ is augmented as follows to obtain $G_{\omega 0}$. Let C be the set of separator nodes of G_ω. A graph that is the union of the shortest path trees T_v in G_ω from each node v in C to the nodes in $C - \{v\}$ is constructed. The induced subgraph of this graph restricted to $G'_{\omega 1}$ is inferred. To keep its size small, this induced subgraph is then contracted by repeatedly replacing each degree 2 node not in C and its incident edges by an edge of cost equal to the sum of the costs of the two edges removed. Graph $G_{\omega 0}$ is the union of the contracted graph and $G'_{\omega 0}$. $G_{\omega 1}$ is similarly obtained from $G'_{\omega 1}$. Note that the nodes of C are considered as part of the augmenting graph. Note also that the augmentation does

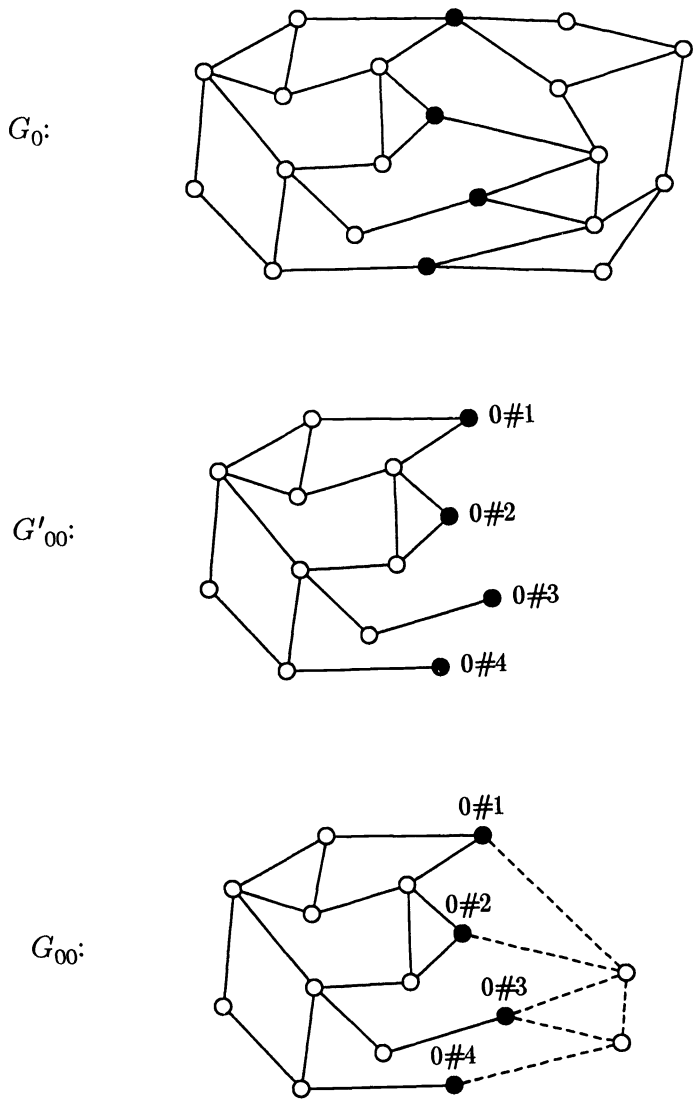


FIG. 1. *Illustration of the first level in the hierarchical decomposition and naming of a c-decomposable graph, with* $c = 4$.

not create any new nodes.

In Fig. 1, the augmenting graph is a subgraph of G'_{01}, with the exception that the dashed edge incident with node 0#1 is the contraction of a path of two edges, each of cost 1, and thus has cost 2.

The following lemmas establish certain important properties of the augmentation.

LEMMA 2.1. *Let G_ω be any graph in the decomposition. The distance in G_ω between any two core nodes u and v is equal to the distance between them in G.*

Proof. The proof is by induction on the length r of ω. The basis $r = 1$ is true since $G_\omega = G_0 = G$. Consider $r > 1$, and suppose that the claim is true for any string of length $r - 1$. Let $G_{\omega'}$ be the graph that is decomposed to produce G_ω, where ω' is a proper prefix of ω and of length $r - 1$. Core nodes u and v of G_ω are core nodes of $G_{\omega'}$ as well and, by the induction hypothesis, the distance in $G_{\omega'}$ between u and v is equal to the distance between them in G.

Consider any shortest (u, v)-path P in $G_{\omega'}$. If P exists in G_ω also, then the lemma follows. Suppose that P does not exist in G_ω. Then each segment of P that is missing in G_ω is a shortest path in $G_{\omega'}$ between a pair of separator nodes of $G_{\omega'}$ and was lost during the separation of $G_{\omega'}$. Due to the augmentation, G_ω contains a path whose length equals the length of the missing segment. Thus, G_ω contains a path of the same length as P, and the lemma follows. □

LEMMA 2.2. *Let the separation of G_ω yield graphs $G'_{\omega 0}$ and $G'_{\omega 1}$. The augmentation of $G'_{\omega 0}$ (respectively, $G'_{\omega 1}$) introduces fewer than c^4 nodes and $3c^4/2$ edges into $G'_{\omega 0}$ (respectively, $G'_{\omega 1}$).*

Proof. We prove the claim for $G'_{\omega 0}$. Let J_0 be the graph with which $G'_{\omega 0}$ is augmented to obtain $G_{\omega 0}$. In worst case J_0 is the contraction of the union of c shortest path trees T_v. There are at most $c(c-1)$ shortest paths in these trees, and for each path there is a corresponding contracted shortest path in J_0. If two shortest paths in J_0 meet, then they share a maximal subpath. We call the endpoints of this subpath *meeting nodes*. In worst case there are $c(c-1)(c(c-1)-1)/2 < c^2(c-1)^2/2$ meetings between different pairs of shortest paths.

We derive an upper bound on the sum of the degrees of the nodes in J_0. Starting with an empty graph, insert the shortest paths of J_0 one at a time. Assign each node a degree when it is introduced into the graph for the first time. Assign it degree 1 if it is in C, and degree 2 otherwise. Taken over all nodes in J_0, this contributes $2(\mid V(J_0) \mid -c) + c = 2 \mid V(J_0) \mid -c$ to the degree sum. If two shortest paths meet, then increase the degree of each of their meeting nodes by 1. Thus the increase in the degree sum due to all meetings between shortest paths is less than $c^2(c-1)^2$. Thus the degree sum is less than $2 \mid V(J_0) \mid -c + c^2(c-1)^2$.

Now each node in $V(J_0) - C$ has degree at least 3, so that the degree sum is at least $3(\mid V(J_0) \mid -c) + c = 3 \mid V(J_0) \mid -2c$. It follows that $\mid V(J_0) \mid -c < c^2(c-1)^2 < c^4$.

The number of edges in J_0 is half the degree sum of J_0. Thus there are fewer than $\mid V(J_0) \mid -c/2 + c^2(c-1)^2/2 < 3c^4/2$ edges in J_0. □

LEMMA 2.3. *Every graph G_ω in the decomposition is c-decomposable.*

Proof. We will show how to find a c-separator of G_ω for any assignment of nonnegative weights to its nodes. View the replacement of each vertex of degree 2 as the contraction of one of its incident edges, followed by the deletion of the resulting loop. The endpoints of the contracted edge can be viewed as identified together, and the resulting vertex can be viewed as a set of vertices identified together. Thus each vertex in G_ω can be viewed as a set of vertices originally in G that have been identified together during the course of the graph decomposition. For each vertex w in G_ω,

choose from its set of vertices a representative vertex and assign this representative a weight (in G) equal to the weight of w in G_ω. For each vertex in G that is not assigned a weight, assign it the weight 0. Since G is c-decomposable, there is a separator C of G, with vertex partition A, B for this assignment of weights. Let C_ω be the set of vertices in G_ω such that a vertex is in C_ω if and only if some member of the vertex's set is in C. Let A_ω, B_ω be a partition of the remaining vertices of G_ω, where a vertex is in A_ω (respectively, B_ω) if and only if the representative of its set is in A (respectively, B).

We claim that C_ω is a c-separator of G_ω, with vertex partition A_ω, B_ω for the given assignment of weights to G_ω. Consider any path P in G between a vertex in A and a vertex in B. Path P must contain some vertex v in C. The corresponding contracted path P_ω (if it exists) in G_ω must contain a vertex in C_ω whose representative in G is v. Since every path in G_ω can be viewed as the contracted version of some path in G, the sets A_ω and B_ω will be nonadjacent. Further, $|C_\omega| \leq |C'| \leq c$ and the total weight of A_ω (respectively, B_ω) is at most the total weight of A (respectively, B), which is at most two-thirds the total weight of G_ω. Thus the claim is true. □

LEMMA 2.4. *Any path P in G between unrelated nodes u and v contains a real ancestor of u and v for some level.*

Proof. Let l be the separating level for u and v and let G_ω be the level $l-1$ graph containing u and v as core nodes. If every node of P is a core node of G_ω, then P exists in G_ω also. Since l is the separating level for u and v, every (u,v)-path in G_ω contains a separator node of G_ω, i.e., an ancestor of u and v for level $l-1$. Further, the separator node on P is a core node of G_ω, i.e., a real ancestor of u and v for level $l-1$. Thus the lemma is true.

If P does not consist entirely of the core nodes of G_ω, then let y be the first noncore node of G_ω encountered on P in going from u to v. Since y is a noncore node of G_ω, there is a smallest prefix ω' of ω such that y is part of the augmentation of $G_{\omega'}$. Further, y is a core node as well as a separator node of $G_{\omega'}$, since it is the first noncore node of G_ω on P. Thus, since u and v are both core nodes of $G_{\omega'}$, it follows that y is a real ancestor of u and v for level l', where l' is the level of $G_{\omega'}$. Thus the lemma is true. □

In particular, Lemma 2.4 implies that any shortest (u,v)-path in G between unrelated nodes u and v contains a real ancestor of u and v for some level. We denote by $real\_ancestor(u,v)$ the first such real ancestor encountered on the shortest path in going from u to v. As we shall see in the next section, $real\_ancestor(u,v)$ plays a crucial role in the routing.

3. Routing information and routing strategy in the basic scheme. Having generated the decomposition and node names, we store appropriate routing information at the nodes and use this to perform the routings. In order to motivate the routing information stored, we first give an overview of the routing strategy.

The strategy for routing from a source s to a destination d depends on whether or not s and d are related. Since it is not expensive to store shortest paths routing information for routing between related nodes, if s and d are related a shortest routing can be performed using this information. However, this approach is not feasible for routing between unrelated nodes, as the amount of shortest paths routing information needed is large. Instead, if s and d are unrelated, then the routing is done as two shortest routings, each of which is between a pair of related nodes. Let a be a suitably chosen ancestor of s and d for level $l-1$, where l is the separating level for s and d. The first routing is from s to a, and the second routing is from a to d. The length of

the overall routing from s to d is not necessarily shortest, but depends on the ancestor chosen. In the basic routing scheme, where the ancestor for level $l-1$ chosen is the one closest to s in G, the routing is within a factor of 3 of optimal. In the enhanced routing scheme described in the next section, we show how to reduce the length of the routings by making a more careful choice of an ancestor.

There are two problems that can arise in the routing strategy described above. The first problem is that the ancestor chosen may not be related to s and d. Thus, shortest paths routing information will not be available to do the routings from s to a and from a to d. The problem is overcome by making the name $real\_ancestor(s,a)$ available to s, since a real ancestor of s and a is clearly a real ancestor of s and d as well. The routing from s to d is done through $real\_ancestor(s,a)$. We call $real\_ancestor(s,a)$ a *surrogate*.

The second problem has to do with routing between related nodes. As an example, consider routing from s to d when they are related. Node s uses its shortest paths information to determine the node w to which the message is to be sent. If w and d are unrelated, then w will be unable to continue the routing to d. The problem is overcome by having s supply the name $real\_ancestor(w,d)$ to w in the message header, so that w can route the message through this node. We call $real\_ancestor(w,d)$ a *milestone* in the routing. In addition, the problem can occur at other intermediate nodes in the routing from s to d, as well as in each phase of the two-phase routing employed when s and d are unrelated. However, each time a suitable milestone will be available through which the routing can be done.

We are now ready to describe in more detail the routing information stored in the basic scheme. Let v be any node. The information at v consists of four tables: $next\_node_v(\cdot)$, $milestone_v(\cdot)$, $ancestor_v(\cdot)$, and $surrogate_v(\cdot)$. The tables $next\_node_v(\cdot)$ and $milestone_v(\cdot)$ are used for routing to related nodes. For each related node u, the name of the next node on a shortest (v,u)-path in G is stored in $next\_node_v(u)$. If $next\_node_v(u)$ and u are unrelated, the name $real\_ancestor(w,u)$ is stored in $milestone_v(u)$, where $w = next\_node_v(u)$. The tables $ancestor_v(\cdot)$ and $surrogate_v(\cdot)$ contain additional information needed for routing to unrelated nodes. Suppose that v is a level i node, $i \geq 1$. For each $j < i$, the name of the ancestor of v for level j that is closest to v in G is stored in $ancestor_v(j)$. If v and $a = ancestor_v(j)$ are unrelated, then the name $real\_ancestor(v,a)$ is stored in $surrogate_v(a)$. For convenience, if v and a are related, then a itself is stored in $surrogate_v(a)$.

The following theorem bounds the amount of routing information stored in the network.

THEOREM 3.1. *For any n-node c-decomposable graph, the basic scheme stores a total of $O(cn\log n)$ items of routing information, where each item is $O(\log n)$ bits long.*

Proof. Each item of routing information is a node name, and hence is $O(\log n)$ bits long.

We first argue that the total amount of information held in $next\_node_v(\cdot)$ by all nodes v is $O(cn\log n)$. Any node v has $O(c\log n)$ real ancestors in the decomposition. Thus v stores $O(c\log n)$ items of information for these ancestors, and these ancestors together store $O(c\log n)$ items of information for v. Therefore, taken over all nodes v, a total of $O(cn\log n)$ items of such information are stored in the network. Further, since each node has at most c siblings, $O(cn)$ items of sibling information are stored in total. The total amount of information held in $milestone_v(\cdot)$ by all nodes v is at most that which is held in $next\_node_v(\cdot)$. Finally, each node v stores $O(\log n)$ items

of information in $ancestor_v(\cdot)$ and $surrogate_v(\cdot)$, for a total of $O(n \log n)$. □

A complete description of the routing from s to d in the basic scheme is as follows. The message header contains separate fields for the milestone and the destination, both initially set to d. The milestone field alone is reset, as necessary, during the routing. Let d' denote the current name in the milestone field. Each node v participating in the routing performs a *routing action*, as follows. It determines $w = next\_node_v(d')$, resets d' to $milestone_v(d')$ if w and d' are unrelated, and then sends the message to w over edge $\{v, w\}$.

At the start of the routing, node s compares the name s with d', which is initially d, to determine whether the two nodes are related or not. If they are related, then s performs a routing action. Otherwise, let l be the separating level for s and d (l can be determined from the names), and let $a = ancestor_s(l-1)$. Then s resets d' to $surrogate_s(a)$ and performs a routing action. Each intermediate node different from the current d' will find the latter in its routing table and thus can perform a routing action. Eventually the message reaches the current d'. If d' is d, then the routing terminates. Otherwise, d' is reset to d and a routing action is performed.

Note that whenever the milestone field is reset at a node that is different from s and the current milestone, it is reset to a real ancestor on a shortest path to the current milestone. Thus it is enough to continue the routing with respect to the new milestone, and the previous milestone need not be saved. (In fact, the message may never even reach some of the milestones that were discarded. This is because the next node information used to do a shortest routing from the current milestone to d might correspond to a shortest path that is different from the shortest path containing some of the discarded milestones.)

In order to establish the performance bound of the basic scheme, we first obtain a lower bound on $\rho(s, d)$ when s and d are unrelated.

LEMMA 3.2. *Let s and d be unrelated nodes with separating level l and let a be the ancestor of s for level $l-1$ that is closest to s in G. Then $\rho(s, d) \geq \rho(s, a)$.*

Proof. We first show that there is a shortest (s, d)-path in G that contains an ancestor of s for level $l-1$. Let G_ω be the level $l-1$ graph containing s and d as core nodes and let P_ω be a shortest (s, d)-path in G_ω. By Lemma 2.1, P_ω has length $\rho(s, d)$. Further, P_ω contains an ancestor b of s and d for level $l-1$, since l is the separating level for s and d. Let G' be the subgraph of G obtained by uncontracting G_ω until no longer possible. This is done by repeatedly taking any edge that represents the contraction of a two-edge path during any augmentation done so far and replacing the edge by the path. Each edge of the two-edge path has the cost it had prior to its contraction. Let P' be the path in G' (and hence in G) that corresponds to the uncontraction of P_ω. P' has the same length as P_ω and contains b. Thus P' is the desired shortest (s, d)-path in G.

Thus, we have

$$\begin{aligned} \rho(s, d) &= \rho(s, b) + \rho(b, d) \\ &\geq \rho(s, b) \\ &\geq \rho(s, a), \end{aligned}$$

since a is the closest ancestor of s for level $l-1$. □

The following theorem establishes the performance bound of the basic scheme.

THEOREM 3.3. *For any c-decomposable graph G, the basic scheme has a performance bound of* 3.

Proof. Let s be any source and d any destination. If s and d are related, then the

routing is along a shortest (s,d)-path in G. This is because every node participating in the routing performs a routing action with respect to the milestone, which is always on a shortest (s,d)-path. Otherwise, let l be the separating level for s and d and a the ancestor of s for level $l-1$ that is closest to it in G. Let $a' = surrogate_s(a)$ and consider the first occasion in the routing that a milestone a'' is reached, where a'' is possibly a'. Since a'' is a real ancestor of s and d, and thus related to both, the routings from s to a'' and from a'' to d are both along shortest paths, by the above reasoning. Thus

$$\begin{aligned}
\hat{\rho}(s,d) &= \rho(s,a'') + \rho(a'',d) \\
&\leq \rho(s,a'') + \rho(a'',a') + \rho(a',d) \\
&= \rho(s,a') + \rho(a',d) \quad \text{(since } a'' \text{ is on a shortest } (s,a')\text{-path)} \\
&\leq \rho(s,a') + \rho(a',a) + \rho(a,d) \\
&= \rho(s,a) + \rho(a,d) \quad \text{(since } a' \text{ is on a shortest } (s,a)\text{-path)} \\
&\leq \rho(s,a) + \rho(a,s) + \rho(s,d) \\
&\leq 3\rho(s,d) \quad \text{(by Lemma 3.2).}
\end{aligned}$$

Thus $\hat{\rho}(s,d)/\rho(s,d) \leq 3$ for any nodes s and d, and the theorem follows. □

In fact, the performance bound of 3 is approachable and is thus the best possible for this scheme. Let $a^\star$ different from a be the ancestor on a shortest (s,d)-path and suppose that $a'' = a' = a$. Let $\rho(s,a) = \rho(s,d) - \rho(a^\star,d)$, and let $\rho(a,d) = \rho(a,s) + \rho(s,d) = 2\rho(s,d) - \rho(a^\star,d)$. Then $\hat{\rho}(s,d)/\rho(s,d) = (3\rho(s,d) - 2\rho(a^\star,d))/\rho(s,d)$ approaches 3 as $\rho(a^\star,d)$ becomes vanishingly small.

4. Improving the performance bound: The enhanced routing scheme. The idea behind the enhanced scheme is to make a more careful choice of an ancestor among the ancestors of s for level $l-1$ when s and d are unrelated. Once a suitable ancestor has been chosen, the routing strategy is as in the basic scheme, with the chosen ancestor substituting for the closest ancestor. To help make the choice, some additional information is stored at the nodes and distance information is encoded in the node names, as follows.

Let v be a level i node, $i \geq 1$. For each $j < i$, instead of storing at v only the name of the closest ancestor of v for level j, we store the names of *all* ancestors u of v for level j. If u and v are unrelated, we also store the name $real\_ancestor(v,u)$ in $surrogate_v(u)$; otherwise we store u in $surrogate_v(u)$. This introduces a total of $O(cn \log n)$ additional items of routing information.

Node v's name is augmented with information about the relative magnitudes of its distances in G from its ancestors for level j. Two pieces of information are encoded for each ancestor, with the information for different ancestors appearing in the lexicographic order of the names assigned to them from the decomposition. The first specifies its position in an ordering of the ancestors by nondecreasing distances from v, with ties broken lexicographically. The second piece of information is as follows. Let $\alpha > 1$ be a function of c to be specified later. For each ancestor a' with index p' in the above ordering by distances, let a'' be the ancestor with the smallest index $p'' > p'$, such that $\rho(v,a') \leq (1/\alpha)\rho(v,a'')$. Then, in addition, p'' is encoded in v's name for a'. If a'' does not exist, then zero is recorded for a'. All this information can be encoded using at most $2c \log c$ bits per level j. As there are at most $\log_{3/2} n = 1.71 \log n$ levels, the total number of additional bits encoded into v's name is $3.42\, c \log c \log n$.

From the ancestors of s and d for level $l-1$, where l is the separating level for s and d, an appropriate ancestor is chosen by s as follows. Clearly, if there are ancestors a' and a'' such that $\rho(s,a') < \rho(s,a'')$ and $\rho(d,a') < \rho(d,a'')$, then a'' can be eliminated. Using the information encoded in its name and that of d, s determines a subset of the ancestors in which no ancestor eliminates another. (These ancestors will be known in terms of their positions in the above lexicographic ordering. However, s can determine their names, since the names of its ancestors at each level are available.) Let $a_1, a_2, \cdots, a_h$ be the $h \leq c$ such ancestors, indexed in increasing order of their distances from s. Denote $\rho(s,a_i)$ by x_i and $\rho(d,a_i)$ by y_i, $1 \leq i \leq h$. Thus $x_1 < x_2 < \cdots < x_h$. Furthermore, since no ancestor eliminates another, we have $y_1 > y_2 > \cdots > y_h$.

Let m be an integer parameter, $1 \leq m \leq h$, to be specified later. If there exists a minimum index i, $1 \leq i < m$, such that $x_i \leq (1/\alpha)x_{i+1}$, then s chooses a_i. Otherwise, if there exists a maximum index i, $m < i \leq h$, such that $y_i \leq (1/\alpha)y_{i-1}$, then s chooses a_i. Otherwise, s chooses a_m. As demonstrated in the proof of the following theorem, the appropriate choice for m is $\lfloor (h+1)/2 \rfloor$.

THEOREM 4.1. *For any c-decomposable graph G, the enhanced routing scheme has a performance bound of $(2/\alpha)+1$, where α, $1 < \alpha \leq 2$, is the root of the equation $\alpha^{\lceil (c+1)/2 \rceil} - \alpha - 2 = 0$.*

Proof. Let s be any source and d any destination. From the proof of Theorem 3.3, we know that the length of the generated routing is at most the sum of the distances in G from s and d to the chosen ancestor. It follows that if there is a shortest (s,d)-path through this ancestor, then the routing is optimal. Thus assume that there is no shortest (s,d)-path through the chosen ancestor, and that there is one through a_q, $1 \leq a_q \leq h$.

Case 1. a_i, $1 \leq i < m$, is chosen in the scan over the x's.

(a) Suppose that $i < q$. Then since $x_{i+1} \leq x_q$ and $x_i \leq (1/\alpha)x_{i+1}$, we have $x_i \leq (1/\alpha)x_q$. Thus,

$$\begin{aligned} \hat{\rho}(s,d)/\rho(s,d) &\leq (x_i + y_i)/(x_q + y_q) \\ &\leq (2x_i + x_q + y_q)/(x_q + y_q) \quad (\text{since } y_i \leq x_i + x_q + y_q) \\ &\leq (2x_q/\alpha)/(x_q + y_q) + 1 \\ &\leq (2/\alpha) + 1. \end{aligned}$$

(b) Suppose that $i > q$. Since $x_j > (1/\alpha)x_{j+1}, 1 \leq j < i$, it can be shown inductively that $x_i < \alpha^{i-q}x_q$. Thus,

$$\begin{aligned} \hat{\rho}(s,d)/\rho(s,d) &\leq (x_i + y_i)/(x_q + y_q) \\ &< (\alpha^{i-q}x_q + y_q)/(x_q + y_q) \quad (\text{from above, and since } y_q > y_i) \\ &\leq \alpha^{i-q} \\ &\leq \alpha^{m-2}. \end{aligned}$$

Case 2. a_i, $m < i \leq h$, is chosen in the scan over the y's.

(a) If $i > q$, then, in a fashion similar to that in Case 1(a), it can be shown that $\hat{\rho}(s,d)/\rho(s,d) \leq (2/\alpha) + 1$.

(b) Suppose that $i < q$. Then $y_i < \alpha^{q-i}y_q$ holds. In a fashion similar to that in Case 1(b), it can be shown that $\hat{\rho}(s,d)/\rho(s,d) < \alpha^{h-m-1}$.

Case 3. a_m is chosen by default.

(a) If $m > q$, then we have $x_m < \alpha^{m-q}x_q$ and, as in Case 1(b), it can be shown that $\hat{\rho}(s,d)/\rho(s,d) < \alpha^{m-1}$.

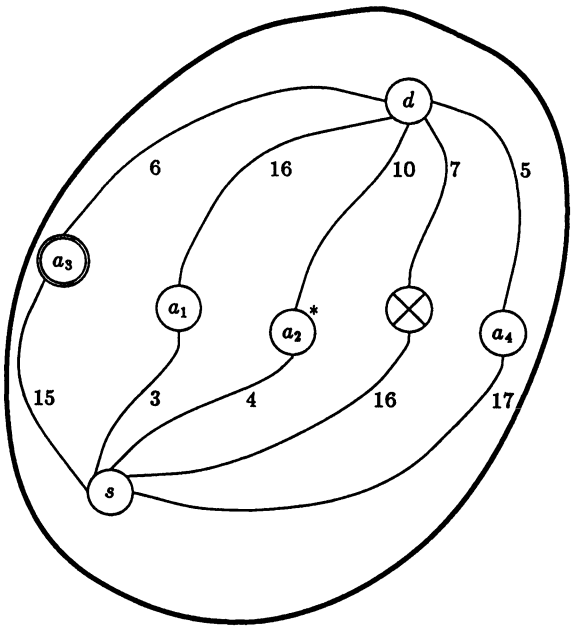


FIG. 2. *Example of a routing choice in the improved scheme for a c-decomposable network, with* $c = 5$.

(b) If $m < q$, then we have $y_m < \alpha^{q-m} y_q$ and, as in Case 1(b), it can be shown that $\hat{\rho}(s,d)/\rho(s,d) < \alpha^{h-m}$.

From the above it follows that

$$\hat{\rho}(s,d)/\rho(s,d) \leq \max\{(2/\alpha)+1, \alpha^{m-1}, \alpha^{h-m}\}.$$

For $m = \lfloor (h+1)/2 \rfloor$ we have $\alpha^{m-1} \leq \alpha^{h-m} \leq \alpha^{\lceil (c-1)/2 \rceil}$. The larger of $(2/\alpha)+1$ and $\alpha^{\lceil (c-1)/2 \rceil}$ is minimized when α is chosen as the positive root of $(2/\alpha)+1 = \alpha^{\lceil (c-1)/2 \rceil}$, i.e., $\alpha^{\lceil (c+1)/2 \rceil} - \alpha - 2 = 0$. Thus $\hat{\rho}(s,d)/\rho(s,d) \leq (2/\alpha)+1$ for any nodes s and d, and the theorem follows. □

For small values of c the above theorem yields performance bounds that are appreciably better than 3. For instance, if c is 2 or 3, then the performance bound is at most 2; if c is 4 or 5, then the performance bound is at most 2.32. These performance bounds are approachable. Let a_1 be the ancestor chosen by s and suppose that there is a shortest (s,d)-path through a_2. Let $x_1 = (1/\alpha)x_2$ and $y_1 = x_1 + x_2 + y_2$. Then $\hat{\rho}(s,d)/\rho(s,d)$ approaches $(2/\alpha)+1$ as y_2 becomes vanishingly small.

Figure 2 illustrates schematically the enhanced routing algorithm for a 5-decomposable graph. There are just four ancestors a_1, a_2, a_3, and a_4 to choose from, since the unnamed ancestor is eliminated by a_3. For this example, $\alpha \approx 1.52$, $m = 2$, and the shortest (s,d)-path is through a_2. In the routing algorithm, the scan over the x's is inconclusive. The scan over the y's first succeeds at y_3, since $y_3 = 6$ and $(1/\alpha)y_2 = 6.6$. Thus a_3 is chosen, yielding a routing that is $21/14 = 1.5$ times longer than optimal.

5. *c*-Separator strategies for graph decomposition. In this section, and in §6, we address the problem of efficiently setting up the routing schemes described. A

crucial step in setting up the schemes is finding a c-separator efficiently. In this section, we give $O(n)$-time c-separator algorithms for two specific classes of c-decomposable graphs, namely, the series-parallel graphs ($c = 2$) and the k-outerplanar graphs for $k > 1$ a constant ($c = 2k$).

5.1. Finding a 2-separator for series-parallel graphs. Two edges in a graph are *series* if they are the only edges incident with a node, and *parallel* if they join the same pair of nodes. A *series-parallel* graph is recursively defined as follows [D]. An edge is a series-parallel graph. The graph obtained by replacing any edge in a series-parallel graph either by two series edges or by two parallel edges is series-parallel. A *two-terminal series-parallel graph* is a graph with two distinguished nodes called *terminals* and is defined recursively as follows. Any edge is a two-terminal series-parallel graph, the terminals being its endpoints. If H_1 and H_2 are two-terminal series-parallel graphs, then so is the graph H obtained either by identifying one of the terminals of H_1 with one of the terminals of H_2 or by identifying them in pairs. In the former case the terminals of H are the unidentified terminals of H_1 and H_2, while in the latter they are the identified terminals. Any simple n-node series-parallel graph has $O(n)$ edges.

With every two-terminal series-parallel graph G, one can associate a *binary structure tree* [VTL]. Each leaf of the tree represents an edge of G. If v is an internal node of the tree with children v_1 and v_2 representing the two-terminal series-parallel graphs H_1 and H_2, then v represents the two-terminal series-parallel graph H obtained as above from H_1 and H_2. The root of the tree represents G. Since every series-parallel graph is two-terminal series-parallel for an appropriate choice of terminals [D], a structure tree can be associated with it.

For convenience, we assume that the given series-parallel graph G is biconnected. This condition can be enforced, if necessary, by introducing an edge between the terminals of G. The cost of the edge is chosen to be greater than the sum of all the edge costs, so that shortest paths are unaffected.

Given any assignment of nonnegative weights to the nodes of G, a 2-separator can be found as follows. Construct a structure tree for G with root r, as described in [VTL]. For each node x in the tree, let $W(x)$ be the sum of the weights of the nodes in the series-parallel graph represented by x. For each nonleaf node, let the *heavy child* be the one with the larger $W(\cdot)$, ties broken arbitrarily.

Initially, set x to r. While x is not a leaf of the structure tree and $W(x)$ exceeds two-thirds the total weight assigned to the nodes of G, reset x to its heavy child. When this step terminates, let C be the set of terminals of the series-parallel graph represented by x. Let A consist of the remaining nodes of this graph and let B be $V(G) - (A \bigcup C)$. It can be verified easily that A, B, and C satisfy the conditions for a 2-separator.

THEOREM 5.1. *A 2-separator of an n-node series-parallel graph can be found in $O(n)$ time.*

Proof. Consider the algorithm described above for finding a 2-separator of a series-parallel graph. The structure tree can be constructed in $O(n)$ time [VTL]. The time to compute $W(\cdot)$ and search the tree is clearly $O(n)$. □

5.2. Finding a $2k$-separator for k-outerplanar graphs. The k-outerplanar graphs are defined as follows [B]. Consider a plane embedding of a planar graph. The nodes on the exterior face are *layer* 1 nodes. For $i > 1$, the *layer* i nodes are those that lie on the exterior face of the embedding resulting from the deletion of all layer j nodes, $j < i$. A plane embedding is *k-outerplane* if it contains no node with

layer number exceeding k. A planar graph is *k-outerplanar* if it has a k-outerplane embedding. Any n-node k-outerplanar graph has $O(n)$ edges.

Let G be a k-outerplanar graph. We assume that a k-outerplane embedding $G^\star$ of G is available. $G^\star$ may be represented using the data structure of [LT], where each node has available a list of its neighbors in clockwise order around the node in the embedding. Given any assignment of nonnegative weights to the nodes of G, a $2k$-separator can be found as follows. The interior faces of $G^\star$ are first triangulated. Each interior face whose boundary consists of nodes all with the same layer number is triangulated arbitrarily. Each interior face whose boundary consists of both layer i and layer $i+1$ nodes, $1 \leq i < k$, is triangulated by repeatedly adding an edge joining a layer $i+1$ node to a layer i node. The resulting embedding, $G^\star_\triangle$, is also k-outerplane, with each layer $i+1$ node adjacent to at least one layer i node. The desired separator is found in $G^\star_\triangle$.

We assume that $G^\star_\triangle$ is biconnected. Otherwise, we enforce this condition as follows. Each articulation point a of $G^\star_\triangle$ will be on the exterior face. Introduce an edge joining two neighbors of a that are on the exterior face and are consecutive in the clockwise ordering of the neighbors of a. The number of such edges introduced will be $O(n)$, and the cost of each is chosen sufficiently large so that shortest paths are unaffected.

The separator algorithm is as follows. At all times, the algorithm maintains a path P of length at most $2k$ in $G^\star_\triangle$, which disconnects $G^\star_\triangle$ into two regions. The algorithm repeatedly modifies P until the total weight of the nodes in each region is at most two-thirds the total weight assigned to the nodes of G. Initially, P consists of a single edge joining a pair of level 1 nodes. In general, P has layer 1 nodes as endpoints, and from one end of P to the other, the layer numbers of its nodes first increase monotonically and then decrease monotonically, possibly with a single pair of consecutive nodes of the same layer number.

For each region bounded by P, determine the sum of the weights of the nodes contained in the region. Let the *heavy region* be the one with the larger total weight, ties broken arbitrarily. If the heavy region has weight exceeding two-thirds the total weight assigned to the nodes of G, then modify P as follows.

Let v be a node on P of highest layer number and u the neighbor of v on P with the higher layer number, ties broken arbitrarily. Let P_1 and P_2 be the subpaths of P on either side of edge $\{v, u\}$, where v is an endpoint of P_1 and u an endpoint of P_2. Consider the face in the heavy region whose boundary contains edge $\{v, u\}$ and let w be the third node on this face. For some i, $1 \leq i < k$, the layer numbers of v, u, and w must each be either i or $i+1$. There are two cases of interest.

If the layer number of w exceeds the layer number of at least one of v and u, then reset P to the path consisting of P_1, $\{v, w\}$, $\{w, u\}$, and P_2. If the heavy region now has total weight exceeding two-thirds the total weight assigned to the nodes of G, then modify P recursively.

Otherwise, let P_3 be a path in the heavy region from w to the exterior face such that the layer numbers of its nodes decrease monotonically. Such a path can be found because each layer $i+1$ node is adjacent to at least one layer i node, $1 \leq i < k$. Furthermore, P_3 can always be picked so that it is either node-disjoint from both P_1 and P_2, or it meets one of these paths at a node and contains the segment of this path from the meeting point to the exterior face. Determine the total weight of the nodes contained in the region R_1 bounded by P_1, $\{v, w\}$, and P_3. Do the same for the region R_2 bounded by P_2, $\{u, w\}$, and P_3. Without loss of generality assume that R_1

is the heavy region. If P_3 and P_1 share no nodes, then reset P to the path consisting of P_1, $\{v, w\}$, and P_3. Otherwise, let z be the first node common to P_1 and P_3 and let e be an edge incident on z from the cycle consisting of the (z, v)-subpath of P_1, the edge $\{v, w\}$, and the (w, z)-subpath of P_3. Reset P to the path consisting of P_1, $\{v, w\}$, and P_3, with e deleted. If the heavy region now has total weight exceeding two-thirds the total weight assigned to the nodes of G, then modify P recursively.

Eventually a path P is found such that the heavy region has total weight at most two-thirds the total weight assigned to the nodes of G. It can be shown inductively that P is a disconnecting path for $G^{\star}_{\triangle}$, hence for $G^{\star}$, and has at most $2k$ nodes. Let C be the set of nodes on P, A be the set of the nodes in the heavy region, and B be $V(G) - (A \bigcup C)$. It may be verified that A, B, and C satisfy the conditions for a $2k$-separator.

THEOREM 5.2. *A $2k$-separator of an n-node k-outerplanar graph can be found in $O(n)$ time.*

Proof. Consider the algorithm described above for finding a $2k$-separator of a k-outerplanar graph. Given the embedding $G^{\star}$ using the data structure of [LT], the layer numbers can be computed in $O(n)$ time [B]. The triangulation can also be done in $O(n)$ time. The time to successively modify paths is as follows. Consider any path P in the algorithm. The node v of highest layer number is identified at the time P is formed. The nodes u and w can be identified in constant time.

If the layer number of w exceeds the layer number of at least one of v and u, then P can then be modified and the weight of the heavy region determined in constant time. The node of highest layer number on the resulting path is w. Charge this cost to edge $\{v, w\}$, which is eliminated from the heavy region. Thus the total time for all paths modified in this fashion is $O(n)$.

Otherwise, we find P and determine as follows which of R_1 and R_2 is the heavy region. Accumulate the weight of the two regions by alternately examining one node from each region, stopping when one of the regions has been exhausted. To do this efficiently, perform a depth-first search in each region in incremental fashion, i.e., search in one region until a node has been added to the depth-first search tree, and then suspend the search in this region and resume it in the other region. Since the graph is planar, the time for this is proportional to the size of the exhausted region. Since the weight of the exhausted region is known, the weight of the other region can be computed, and the heavy region determined. P is then reset appropriately. The node of highest layer number on the resulting path is one of v, u, and w.

The time to thus modify P is proportional to the size of the exhausted region. Charge this cost to the nodes in the region that is not the heavy region. This results in constant charge per node. Since each of these nodes is charged at most once and then eliminated, the total time for all paths modified this way is $O(n)$. □

6. Computing the node names and routing information. In this section we discuss how to generate the node names and determine the routing information stored at each node. Our time bounds hold for any uniformly sparse and contractible class of c-decomposable graphs such that any n-node graph from the class has an $O(n)$-time c-separator algorithm. We call a class of graphs *uniformly sparse and contractible* if, for any graph in the class, the number of edges in any subgraph is linear in the number of nodes, and any contraction of the subgraph is also in the class. Examples of uniformly sparse and contractible classes of c-decomposable graphs are the series-parallel graphs and the k-outerplanar graphs, for $k > 1$ a constant. Linear-time c-separator algorithms for these classes have been given in the previous section.

The following theorem establishes the time needed to set up our routing schemes.

THEOREM 6.1. *Let G be any n-node graph drawn from a uniformly sparse and contractible class of c-decomposable graphs such that any n-node graph from the class has an $O(n)$-time c-separator algorithm. The basic and enhanced routing schemes can be implemented in G in $O(cn(\log n)^2 + c^4 n \log n)$ time. If G is also planar, then the time is $O(cn(\log n)^{3/2} + c^2 n \log n + c^4 n(\log n)^{1/2})$. For series-parallel graphs G, the setup time is $O(n \log n)$.*

Proof. Let $NC(n)$ be the total number of noncore nodes that are generated when graph G with n (core) nodes is decomposed down to graphs with at most c core nodes, where we include in $NC(n)$ each occurrence of a node as a noncore node. We establish an upper bound on $NC(n)$, which will be useful later. We have,

$$\begin{aligned} NC(n) &= 0 \quad \text{for } n \le c \\ NC(n) &< NC(an) + NC((1-a)n) + 2c^4 \quad \text{for } n > c, \end{aligned}$$

where $1/3 \le a \le 2/3$. The last line above follows from the fact that the two graphs resulting from the separation of G have at most an and $(1-a)n$ core nodes, respectively, for some a, $1/3 \le a \le 2/3$. By Lemma 2.2, the augmentation introduces fewer than c^4 noncore nodes into each of these graphs, which contributes a total of at most $2c^4$ to $NC(n)$.

An induction on n shows that $NC(n) \le \max\{0, 2c^3 n - 2c^4\}$ for all $n \ge 0$. Thus $NC(n)$ is $O(c^3 n)$.

We first analyze the setup time for the basic scheme. The time for doing the decomposition and naming is as follows. Let G_ω be any level i graph in the decomposition, $i \ge 0$. Let G_ω have l_ω core nodes and m_ω noncore nodes, for a total of n_ω nodes. Note that G_ω will be in the same class as G. The time to separate G_ω, if necessary, is $O(n_\omega)$. The time to augment the two graphs resulting from the separation of G_ω is dominated by the cost of computing at most c shortest path trees, one rooted at each separator node of G_ω. This takes time $O(cn_\omega \log n_\omega)$, which is $O(cn_\omega \log n)$, using the algorithm from [J]. The other operations, including the contraction, take $O(n_\omega)$ time. The time to name the level i nodes from the core of G_ω is $O(c)$. Thus the time to handle G_ω is $O(cn_\omega \log n)$, which is $O(c(l_\omega + m_\omega)\log n)$. The total time is obtained by summing over all level i graphs and then summing over all levels i. Thus the total time is $O(\sum_{\text{all levels } i} \sum_{\text{all level } i \text{ graphs } G_\omega} c(l_\omega + m_\omega)\log n)$, which is

$$O(\textstyle\sum_{\text{all levels } i}(cn \log n + c \log n \sum_{\text{all level } i \text{ graphs } G_\omega} m_\omega)).$$

This follows, since the cores of the different level i graphs are disjoint, so that

$$\textstyle\sum_{\text{all level } i \text{ graphs } G_\omega} l_\omega \text{ is } O(n).$$

Since

$$\textstyle\sum_{\text{all levels } i} \sum_{\text{all level } i \text{ graphs } G_\omega} m_\omega \text{ is } NC(n),$$

and since there are $O(\log n)$ levels, the total time is $O(cn(\log n)^2 + c^4 n \log n)$.

Next, routing information is set up at the nodes. Let v be a level i node from the core of level i graph G_ω. The $next\_node_v(\cdot)$ information is determined as follows. A shortest path tree rooted at v is constructed in G_ω. Let u be any node in the core of G_ω, and let y be the the child of v on the path from v to u in the tree. If the edge $\{v, y\}$ belongs to G, then $next\_node_v(u)$ is set equal to y. If $\{v, y\}$ does not belong to G, then it must be the result of contracting a maximal path in G, with interior nodes of degree two, during some augmentation. In this case, $next\_node_v(u)$ is set equal to the name of the neighbor of v on this path. This neighbor information can be maintained easily during the augmentations. Furthermore, whenever v is included

in the shortest path tree rooted at a real ancestor w of v for some level less than i, $next\_node_v(w)$ is set equal to the name of the parent of v in the tree, with contracted edges handled as before. During this computation, $\rho(v, w)$ is also determined for later use.

The time to construct the shortest path tree for v is $O(n_\omega \log n_\omega)$. Thus the time to construct the shortest path trees for all level i nodes from the core of G_ω is $O(cn_\omega \log n_\omega)$, which is $O(cn_\omega \log n)$. The total time is obtained by summing over all level i graphs and then summing over all levels i. By a previous argument, this is $O(cn(\log n)^2 + c^4 n \log n)$.

The closest ancestors are determined next. Let $j = 0, 1, 2, \cdots$ in turn and for each j let $i = j+1, j+2, \cdots$ in turn. For each level i node v, the closest ancestor for each level j is determined as follows. Let a be any ancestor of v for level j. If v and a are related, then $\rho(v, a)$ is known from the $next\_node_v(a)$ computation done previously. If v and a are unrelated, then $\rho(v, a)$ can be computed by minimizing $\rho(v, a') + \rho(a', a)$ over all ancestors a' of v and a for level $j' - 1$, where $j' \leq j$ is the separating level for v and a. Since $j' - 1 < j$, the distances $\rho(v, a')$ and $\rho(a', a) = \rho(a, a')$ will have been computed already. If $\hat{a}$ is found to be the closest ancestor of v for level j, then the name $\hat{a}$ is stored in $ancestor_v(j)$.

The above process also yields $real\_ancestor(v, a)$ when v and a are unrelated. Let a' be the ancestor of v and a for level $j' - 1$ that minimizes $\rho(v, a') + \rho(a', a)$. If a' is a real ancestor of v and a for some level, then $real\_ancestor(v, a)$ is a'. Otherwise, $real\_ancestor(v, a)$ is just $real\_ancestor(v, a')$, and the latter will already be available, since $j' - 1 < j$. Thus, corresponding to $\hat{a}$, the name $real\_ancestor(v, \hat{a})$ is stored in $surrogate_v(\hat{a})$ if $\hat{a}$ is a pseudo-ancestor of v for level j. Otherwise, $\hat{a}$ itself is stored in $surrogate_v(\hat{a})$.

The time to compute $\rho(v, a)$ for unrelated nodes v and a is $O(c)$. Thus the time for all such nodes a among the ancestors of v for level j is $O(c^2)$. Once the distances from v to all its ancestors for level j are known, the closest ancestor and the corresponding surrogate can be found in $O(c)$ time. Thus the time for closest ancestor and surrogate computations for all levels at v is $O(c^2 \log n)$, hence $O(c^2 n \log n)$ at all nodes.

The milestone information can be set up as follows. Let v be any level i node from the core of level i graph G_ω. Let u be a core node of G_ω, and thus related to v, and suppose that $w = next\_node_v(u)$ and u are unrelated. Let j be the separating level for w and u, and y the ancestor of u and w for level j that minimizes $\rho(w, y) + \rho(y, u)$. Then $milestone_v(u)$ is just $real\_ancestor(w, y)$, which, by the previous discussion, will be known already. If $next\_node_u(v)$ and v are unrelated, then $milestone_u(v)$ can be set up simultaneously at u.

The time to set up $milestone_v(u)$ is $O(c)$ for each node u in the core of G_ω, and hence $O(cl_\omega)$ for all u in the core of G_ω. Since there are $O(c)$ level i nodes from the core of G_ω, the milestones at all these nodes can be set up in $O(c^2 l_\omega)$ time. The total time to set up milestone information at all level i nodes in the decomposition, which is obtained by summing over all level i graphs G_ω, is $O(c^2 n)$. Thus the total time, obtained by summing over all levels i, is $O(c^2 n \log n)$.

It follows that the total setup time for the basic scheme is $O(cn(\log n)^2 + c^4 n \log n)$. For planar graphs, the faster algorithm from [F] may be used in lieu of the algorithm from [J] for determining shortest paths. This leads to a setup time of $O(cn(\log n)^{3/2} + c^2 n \log n + c^4 n(\log n)^{1/2})$. For series-parallel graphs, a setup time of $O(n \log n)$ can be achieved, using a result from [HT] which allows single-source shortest paths to be computed in $O(n)$ time.

The analysis for the enhanced scheme is as follows. The time for doing the decomposition and naming, and for setting up shortest paths information, milestones, and surrogates, is as before. The time to encode the additional information into the name of a level i node v is as follows. A lexicographic ordering of all the nodes, based on the names assigned from the decomposition, can be generated in $O(n \log n)$ time using a radix sort. A lexicographic ordering of the ancestors of v for any level $j < i$, which number at most c, can be inferred from the full lexicographic ordering in $O(c \log c)$ time, by sorting the positions of these ancestors in the full ordering. Since the distances from v to the ancestors are known, the distance ordering can be generated and the corresponding position information encoded into v's name in $O(c \log c)$ time. Furthermore, given α, the information about the relative magnitudes of distances can be determined in $O(c)$ time, by scanning in increasing order the distances of v from its ancestors for level j. This information can then be encoded in v's name in $O(c \log c)$ time. Thus, the time per level for v is $O(c \log c)$, hence $O(c \log c \log n)$ for all levels. Taken over all nodes, this is $O((c \log c) n \log n)$.

Thus the overall setup time for the enhanced scheme is $O(cn(\log n)^2 + c^4 n \log n)$. It is $O(cn(\log n)^{3/2} + c^2 n \log n + c^4 n (\log n)^{1/2})$ for planar networks, and $O(n \log n)$ for series-parallel networks. □

7. Acknowledgments. We would like to thank the referees for numerous suggestions that helped improve the paper.

REFERENCES

[B] B. S. BAKER, *Approximation algorithms for NP-complete problems on planar graphs*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, Tucson, AZ, October 1983, pp. 265–273.

[D] R. J. DUFFIN, *Topology of series-parallel networks*, J. Math. Appl., 10 (1965), pp. 303–318.

[F] G. N. FREDERICKSON, *Fast algorithms for shortest paths in planar graphs, with applications*, SIAM J. Comput., 16 (1987), pp. 1004–1022.

[FJ1] G. N. FREDERICKSON AND R. JANARDAN, *Designing networks with compact routing tables*, Algorithmica, 3 (1988), pp. 171–190.

[FJ2] ———, *Efficient message routing in planar networks*, SIAM J. Comput., 18 (1989), pp. 843–857.

[H] F. HARARY, *Graph Theory*, Addison-Wesley, Reading, MA, 1969.

[HT] R. HASSIN AND A. TAMIR, *Efficient algorithms for optimization and selection on series-parallel graphs*, SIAM J. Algebraic Discrete Methods, 7 (1986), pp. 379–389.

[J] D. B. JOHNSON, *Efficient algorithms for shortest paths in sparse networks*, J. Assoc. Comput. Mach., 24 (1977), pp. 1–13.

[KK] L. KLEINROCK AND F. KAMOUN, *Hierarchical routing for large networks – performance evaluation and optimization*, Comput. Networks, ISDN Systems, 1 (1977), pp. 155–174.

[LT] R. J. LIPTON AND R. E. TARJAN, *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36 (1979), pp. 177–189.

[M] G. MILLER, *Finding small simple cycle separators for 2-connected planar graphs*, J. Comput. System Sci., 32 (1986), pp. 265–279.

[PU] D. PELEG AND E. UPFAL, *A trade-off between space and efficiency for routing tables*, J. Assoc. Comput. Mach., 36 (1989), pp. 510–530.

[SK] N. SANTORO AND R. KHATIB, *Labelling and implicit routing in networks*, Comput. J., 28 (1985), pp. 5–8.

[VTL] J. VALDES, R. E. TARJAN, AND E. L. LAWLER, *The recognition of series-parallel digraphs*, SIAM J. Comput., 11 (1982), pp. 298–313.

[vLT1] J. VAN LEEUWEN AND R. B. TAN, *Computer networks with compact routing tables*, in The Book of L, G. Rozenberg and A. Salomaa, eds., Springer-Verlag, Berlin, New York, 1986, pp. 259–273.

[vLT2] ———, *Interval routing*, Comput. J., 30 (1987), pp. 298–307.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 182–204, February 1990

012

FEASIBLE REAL FUNCTIONS AND ARITHMETIC CIRCUITS*

H. JAMES HOOVER†

Abstract. The connection between computable analysis and computational complexity is investigated by asking what it means to feasibly compute a real function. A new class of arithmetic circuits, called feasible-size-magnitude, is introduced and used to show a feasible version of the Weierstrass approximation theorem. That is, a real function is feasible if and only if it can be sup-approximated by a division-free uniform family of feasible-size-magnitude arithmetic circuits over R. This result involves a counter-intuitive simulation of Boolean circuits by arithmetic ones. It also has implications for algebraic complexity theory.

Key words. feasible analysis, arithmetic circuit complexity, computable analysis

AMS(MOS) subject classifications. 68Q05, 26C99, 41A10, 41A20

0. Introduction.

Suppose that x is a real number, and that f is a continuous function over the real interval $(-\infty, +\infty)$. What does it mean to compute x, to compute $f(x)$ for any x, and to do so efficiently or feasibly?

The computability aspect of this question originates with Turing [Tu36],[Tu37], with further work by Grzegorczyk [Gr57] and Shepherdson [Sh76] among many others. The notions of feasible real number and feasible real function are more recent, being established by Ko and Friedman in [KF82]. They define a real number x to be feasible if an approximation to x of error $\leq 2^{-n}$ can be computed in time $n^{O(1)}$. Similarly, a real function f is feasible over the fixed interval $[0, 1]$ if an approximation to $f(x)$ of error $\leq 2^{-n}$ can be computed in time $n^{O(1)}$ relative to the cost of computing approximations to x. In other words, a real function is a reduction of each approximation of $f(x)$ to a set of oracle calls delivering approximations to the input x. Thus, real numbers are computed by Turing machines, while functions are computed by *oracle* Turing machines.

From a classical real analysis perspective, it is sufficient to study only the functions defined on the interval $[0, 1]$. However this is not the case if one adds complexity considerations. In [Ho87] we began a systematic study of analysis in the more general case, beginning with real numbers, progressing to functions, and then considering operators. This paper contains some of the results of this program:

- First we extend the notion of feasible function to the interval $(-\infty, +\infty)$ by making the complexity of the function f depend on both the desired accuracy of approximation and the length of the interval over which it is computed. The structure of the oracle machine computation in the case of continuous functions is such that, although one can make many oracle calls, only two are actually needed. This allows us to give equivalent definitions of feasible real functions in terms of the simpler Boolean circuit model without oracles.
- Both the oracle machine and Boolean circuit models are unstructured in the sense that they are permitted to inspect the bits of their inputs and to modify their behaviour accordingly. This is not quite how mathematicians compute functions. Approximation theorists use more structured ways—such as power series, polynomials,

* Received by the editors September 8, 1988; accepted for publication (in revised form) May 9, 1989. This research was supported by the Natural Sciences and Engineering Research Council of Canada grant OGP 38937.

† Department of Computing Science, University of Alberta, Edmonton, Alberta, Canada T6G 2H1 (hoover@cs.ualberta.ca).

and rational functions—which are essentially oblivious to the actual values of their inputs. We show that there is common ground between these two approaches.

- To capture the notion of structured computations, we introduce a new class of arithmetic circuits, called *feasible-size-magnitude*, which are of polynomial size and in which internal gate values have only polynomial length magnitudes. This new model is more like what an approximation theorist would consider a legitimate computation.
- Although one might think that the unstructured model is more powerful than the structured one, our main result is that they are equivalent. That is, a real function f is feasible on an oracle Turing machine if and only if it can be approximated by a uniform family of feasible-size-magnitude arithmetic circuits. In fact, the approximating family is inverse-free and thus simply computes a polynomial—a kind of feasible Weierstrass approximation theorem.
- An interesting consequence of the preceding result is that an analogue of Strassen's [Str73] division removal result also holds for feasible-size-magnitude arithmetic circuits. Specifically, any feasible-size-magnitude circuit family can be approximated by an inverse-free feasible-size-magnitude circuit family. This is despite the fact that such circuit families can have exponential degree and do not necessarily compute polynomials—two conditions under which Strassen's technique fails.
- Although these results are in the domain of real analysis, they have algebraic applications. We show that given a feasible-size-magnitude circuit family computing a polynomial, it is in general impossible to produce a feasible-size-magnitude circuit family that computes the indefinite integral of that polynomial unless $P = \#P$. This partly addresses an open problem posed by Kaltofen in [Ka87].
- We also extend these notions to feasible space and give corresponding results for oracle machine space versus arithmetic circuit depth.

We assume that the reader is familiar with the usual basic material of computational complexity, Turing machines, uniform Boolean and arithmetic circuits, and real analysis.

1. Feasible reals. Since most real numbers cannot be represented as finite strings of digits, any notion of computing a real number must involve approximation.

DEFINITION 1.1. Let x be a real number. The notation $\langle x \rangle_n$, for $n \geq 0$, stands for any rational number such that $|x - \langle x \rangle_n| \leq 2^{-n}$. We say that $\langle x \rangle_n$ is an nth *approximation* to x, and that a sequence $\{\langle x \rangle_n\}$ of such approximations *represents* x.

Often, one wishes to perform arithmetic on approximations. For example, suppose one wants an nth approximation to $x + y$. What approximations to x and y are sufficient? Equivalently, suppose one has a representation of x and y, what is a representation of $x + y$?

PROPOSITION 1.2. *Let x and y be real numbers represented by $\{\langle x \rangle_n\}$, $\{\langle y \rangle_n\}$. Also suppose that $L \geq 0$ and $M \geq 0$ are integers such that $2^{-L} \leq |x| \leq 2^M$ and $|y| \leq 2^M$. Then:*

1. *$x+y$ can be represented by the sequence with terms $\langle x+y \rangle_n \equiv \langle x \rangle_{n+1} + \langle y \rangle_{n+1}$.*
2. *xy can be represented by $\langle xy \rangle_n \equiv \langle x \rangle_{n+M+2} \langle y \rangle_{n+M+2}$.*
3. *x^{-1} can be represented by $\langle x^{-1} \rangle_n \equiv \langle x \rangle_{n+3L}^{-1}$.*

The natural notion of feasibility for real numbers is that computing an n-bit approximation to x should only require time $n^{O(1)}$.

DEFINITION 1.3. A real number x is a *feasible real* if there is a Turing machine that, on input of a natural number n, outputs $\langle x \rangle_n$ in time $n^{O(1)}$.

The results that follow depend in a detailed way on how we encode rational numbers and sequences of approximations. To simplify the exposition we will restrict

our rationals to those that can be encoded in fixed-point binary notation, and restrict our representations so that every nth approximation to x has exactly n bits to the right of the binary point. This does not affect the class of reals and real functions that we wish to compute—although this requires some proof. (See [Ho87].)

DEFINITION 1.4. Let l be a natural number, s an integer, and $b_0, \cdots, b_l \in \{0, 1\}$. Consider the rational x where $x = (2b_0 - 1)2^s \sum_{i=1}^{l} 2^{-i} b_i$. We say that $[s, l, b_0, \cdots, b_l]$ is a *range s length l fixed-point binary encoding of x*. Note that b_0 plays the role of a sign bit.

DEFINITION 1.5. Let s be a fixed integer, and x a real in $[-2^s, 2^s]$. We say that a sequence of approximations $\{\langle x \rangle_n\}$ is a *range s fixed-point representation of x* if every $\langle x \rangle_n$ is encoded as a range s length l fixed-point binary encoding with $l = s + n$.

Thus we compute a real x by computing a sequence $\{\langle x \rangle_n\}$ of approximations, each term having essentially one more bit of precision than the preceding one.

2. Feasible real functions. The original work of Ko and Friedman [KF82] defines the class of feasible real functions in terms of oracle Turing machines.

DEFINITION 2.1. An *oracle machine M on sequence ω* is a multitape Turing machine with input and output tapes and with two other distinguished tapes. One tape, called the *oracle index tape*, is write only. The other, called the *oracle result tape*, is read only. When a natural number n is written on the oracle index tape the string $\langle \omega \rangle_n$ may be read from the oracle result tape. Such an operation is termed an *oracle call*. The time complexity of M is the usual one for Turing machines, with the cost of each oracle call being the length of n and $\langle \omega \rangle_n$.

Oracle machines are the most powerful of all the reasonable deterministic models for computing real functions. In the most general case, to compute an approximation $\langle f(x) \rangle_n$ we let the machine make any number of oracle calls to obtain various approximations to x, and perform arbitrary amounts of computation between calls. In general, every computable function is continuous on its domain [Gr57]. We are interested in those functions whose domain is the entire real line.

DEFINITION 2.2. Oracle Turing machine M *computes* a continuous real function f on the interval $(-\infty, +\infty)$ if for each integer $s \geq 0$, for each real $x \in [-2^s, 2^s]$, and for every oracle sequence $\{\langle x \rangle_n\}$ that is a range s fixed-point representation of x, machine M, on input $n \geq 0$, outputs $\langle f(x) \rangle_n$.

What should our notion of feasibility be for real functions? We must use a weaker notion than do Ko and Friedman. They require that one function $t(n) = n^{O(1)}$ work for all input sequences $\{\langle x \rangle_n\}$, so that M on input n outputs $\langle f(x) \rangle_n$ in time $t(n)$. This stronger notion can be used only when the length of the string of bits encoding $\langle x \rangle_n$ that result from an oracle call of n is independent of x—such as when the sequence represents reals only in $[0, 1]$. If one wishes to compute functions over $(-\infty, +\infty)$ as we do, we must accommodate arbitrary reals, and so the length of $\langle x \rangle_n$ will increase as the real it represents increases. In this case, no fixed time bound t will cover all inputs. The solution is to make the notion of feasibility also sensitive to the magnitude of the input argument.

DEFINITION 2.3. Real function f is *feasible* if there is an oracle machine M computing f such that for every real x, on input of oracle sequence $\{\langle x \rangle_n\}$ of range s representing x, oracle machine M on input n outputs $\langle f(x) \rangle_n$ in time $n^{O(1)} s^{O(1)}$.

Under this definition of feasibility, most of the usual mathematical functions such as min, max, $\sqrt{x}$, sin, and cos and their inverses are feasible real functions. The exponential function e^x is not however, because its magnitude grows too quickly as a function of x. But the restricted version $e^{\min\{x,a\}}$, for any fixed a, is feasible.

It is an instructive exercise to pick a function and show that it is feasible. For example, this function is feasible:

$$w(x) \equiv \begin{cases} e^{-1/(1-x^2)} & \text{if } x \in (-1,1) \\ 0 & \text{otherwise} \end{cases}$$

It is also illuminating to think about why $1/x$ is not feasible. (But the function $1/\max\{|x|, a\}$ is feasible for fixed nonzero a.)

In the most general setting, oracle Turing machines seem to be necessary for computing real functions. But when resource bounds are applied to the computations a simpler model will suffice, and one does not need the full power of the oracle model. This is because oracles that represent reals have a special property—oracle call $\langle x \rangle_i$ contains essentially all the information provided by oracle calls $\langle x \rangle_0$ to $\langle x \rangle_{i-1}$. In any computation with a given n and s, the machine can make oracle calls with index at most $n^{O(1)}s^{O(1)}$. So for each particular oracle, one call to get $\langle x \rangle_0$ will determine s (by just counting the bits in $\langle x \rangle_0$), and then one call with index $n^{O(1)}s^{O(1)}$ can be used for all the other calls made by the machine. Thus the oracle machine need only make two oracle calls to compute an approximation to $f(x)$. The net effect is that we do not require the oracle mechanism at all and instead can express the computation in terms of Turing machines that take as input a single approximation of x. For further details see [Ho87].

This observation lets us describe the computation of feasible functions in terms of the more natural model of uniform Boolean circuits. There are many notions of uniformity (see [BCH86],[Bo77]). For our purposes, a circuit family $\{\gamma_n\}$ is *uniform* if a description of member γ_n can be produced by a Turing machine using space $O(\log \text{size}(\gamma_n))$.

The idea is to define a uniform circuit family $\{\gamma_n\}$ that over the interval $[-2^n, 2^n]$ computes an approximation $\langle f(x) \rangle_n$ to $f(x)$. Since we only want to approximate f we need not have x exactly, and since the interval of approximation is closed, with the function f continuous, there will be some upper bound on how accurately we need to approximate x. There will also be an upper bound on the magnitude of f itself over the interval $[-2^n, 2^n]$. Both of these upper bounds are functions of n, and are called the *modulus* and *range* functions respectively. Their presence is a crucial part of the definition of the family $\{\gamma_n\}$ since each member circuit can only have a fixed number of input and output bits.

Each circuit γ_n takes as input a range n length $n+\mu(n)$ fixed-point binary encoding of an approximation $\langle x \rangle_{\mu(n)}$. It delivers a range $\rho(n)$ length $\rho(n)+n$ fixed-point binary encoding of an approximation $\langle f(x) \rangle_n$, which we denote by $\gamma_n(\langle x \rangle_{\mu(n)})$. The situation is depicted in Fig. 2.1, where the bits b_i are from the fixed-point representation of $\langle x \rangle_{\mu(n)}$ and the bits c_i are from the fixed point representation of $\langle f(x) \rangle_n$. The equivalence of the two notions of feasibility is summarised by the following theorem:

THEOREM 2.4. *A real function f is feasible if and only if there is a uniform family $\{\gamma_n\}$ of Boolean circuits with* $\text{size}(\gamma_n) = n^{O(1)}$*; a modulus function μ from naturals to naturals; and a range function ρ from naturals to naturals such that when $\langle x \rangle_{\mu(n)}$ is a range n length $n+\mu(n)$ fixed-point binary encoding then $\gamma_n(\langle x \rangle_{\mu(n)})$ is a range $\rho(n)$ length $\rho(n)+n$ fixed-point binary encoding, and $|f(x) - \gamma_n(\langle x \rangle_{\mu(n)})| \leq 2^{-n}$. Note that $\mu(n) = n^{O(1)}$, $\rho(n) = n^{O(1)}$ and $\gamma_n(\langle x \rangle_{\mu(n)}) \in [-2^{\rho(n)}, 2^{\rho(n)}]$.*

Proof. The proof simply uses the standard equivalences between Turing machines and uniform Boolean circuit families as described in [Bo77]. □

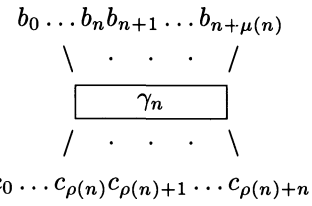


FIG. 2.1. *Boolean circuit γ_n approximating real function f.*

The following lemma is justification for calling μ a modulus function—as in modulus of continuity.

LEMMA 2.5. *Let f be a feasible real function computed by Boolean circuit family $\{\gamma_n\}$ with modulus function μ and range function ρ. Then for all $x, x' \in [-2^n, 2^n]$, if $|x - x'| \leq 2^{-\mu(n)}$ then $|f(x) - f(x')| \leq 2^{-n+1}$.*

Proof. Since $|x - x'| \leq 2^{-\mu(n)}$ there exists a range n length $n + \mu(n)$ fixed-point binary number z such that $|z - x| \leq 2^{-\mu(n)}$ and $|z - x'| \leq 2^{-\mu(n)}$. That is, $z \equiv (2b_0 - 1)2^n \sum_{i=1}^{n+\mu(n)} 2^{-i} b_i$ is an $\mu(n)$th approximation to both x and x'. Thus $\gamma_n(z)$ is an nth approximation to both $f(x)$ and $f(x')$, and so we have that $|f(x) - f(x')| \leq 2^{-n+1}$. □

These definitions can be extended in the obvious way to functions that have multiple input and output arguments.

3. Feasible-size-magnitude arithmetic circuits. The preceding notions reflect an unstructured approach to computing real functions in the sense that what happens inside the oracle Turing machine or Boolean circuit family is completely arbitrary, generally involving decisions and possibly even strange bit computations.

The main problem with using an unstructured model to compute real functions occurs when one wishes to supply a function to an operator. For example, suppose that one wishes to compute $\int_0^x f(y)dy$ for arbitrary functions f. When f is presented as a black-box that does some kind of obscure bit manipulations, the only way to compute the integral is by taking samples of f and integrating numerically. But if f is presented as a polynomial, then the structure in the description of f can be exploited and f can be integrated symbolically.

One structured approach to computing a real function is to use *uniform arithmetic circuits* to approximate the function. These circuits do nothing but arithmetic operations, and are the natural extension to the traditional notions of approximation by polynomials and rational functions [BB88]. On the surface, this approach appears to be weaker than the unstructured one. In fact, the original motivation for using arithmetic circuits was to get a computationally weaker model. What is rather surprising is that the Boolean circuit model is equivalent to the arithmetic circuit model when computing feasible real functions.

Uniform arithmetic circuits over the reals R are acyclic networks of gates where the edges carry real numbers and the gates perform the operations $+$, $-$, $\times$, $\cdot^{-1}$ (inverse) or deliver rational constants. A computation by such a circuit is the obvious one, with the proviso that the computation is undefined when any inverse gate has a zero input. These circuits and their extensions to general fields have been extensively investigated, [vzG87], [vzGS86], and are one of the main models of parallel algebraic complexity. The notion of uniformity can be made precise for these types of circuits,

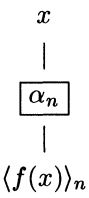


FIG. 3.1. *Arithmetic circuit α_n approximating real function f.*

but for our purposes the following informal definition will suffice:

DEFINITION 3.1. An arithmetic circuit family $\{\alpha_n\}$ is *log-space uniform* if a description of the connection pattern, gate types, and values of the constant gates (encoded in binary) for circuit α_n can be produced in space $O(\log n)$ on a deterministic Turing machine.

Each arithmetic circuit computes a rational function over R. In general, a family of arithmetic circuits $\{\alpha_n\}$ computes a sequence of rational functions and one can ask about approximating a function f with such a sequence. It is important to note that for many f no single rational function can approximate f within 2^{-n} over the entire interval $(-\infty, +\infty)$. For example, the function $\sin(x)$ has an infinite number of zeros, so any rational function $P(x)/Q(x)$ that is within 2^{-n}, $n \geq 2$, of $\sin(x)$ must also have an infinite number of zeros, which implies that $P(x)$ is either constant or has infinite degree. So any approximating family of rational functions will require an index that specifies the range over which the approximation works.

We use one index to indicate both the accuracy of approximation and the range over which it applies. Each circuit α_n of the approximating family $\{\alpha_n\}$ for f takes as input a real $x \in [-2^n, 2^n]$, and computes a real output, denoted by $\alpha_n(x)$, which approximates $f(x)$. Pictorially we have Fig. 3.1, where we slightly abuse the notation $\langle f(x) \rangle_n$ and allow it to denote real values, not just rational ones. Note that if x is a rational number, then $\alpha_n(x)$ will also be a rational number.

We can now define what it means to compute a real function with arithmetic circuits.

DEFINITION 3.2. Let $\{\alpha_n\}$ be a uniform family of arithmetic circuits over R, and let f be a real function. Suppose that for all $n \geq 0$ circuit α_n satisfies the relation that if $x \in [-2^n, 2^n]$ then $|f(x) - \alpha_n(x)| \leq 2^{-n}$. Then we say that the family $\{\alpha_n\}$ of arithmetic circuits *sup-approximates* real function f.

What does it mean to do this feasibly? Algorithms in algebraic complexity theory are often sensitive to the degree of the circuit, and so in order to maintain feasibility one usually limits the degree of the function computed by α_n to $n^{O(1)}$ (see [Va82]). But there are functions, such as sin, that require exponential degree to approximate, yet are feasible in the unstructured sense. Since we wish to compute such functions, rather than limiting degree, we instead place a bound directly on the magnitude of the rational numbers produced in the circuit. Note that a degree bound implies a magnitude bound so we are relaxing the usual constraint. Also note that magnitude bounds make sense in the general algebraic setting only when there is some norm that measures the field elements.

With the above considerations in mind we give the following definitions.

DEFINITION 3.3. Let $\{\alpha_n\}$ be an arithmetic circuit family over R, and let $\alpha_n^v(x)$

denote the output value of gate v of α_n on input x. The *magnitude* of circuit α_n, denoted $\mathrm{mag}(\alpha_n)$ is the quantity

$$\mathrm{mag}(\alpha_n) \equiv \max_{v \in \alpha_n} \max_{x \in [-2^n, 2^n]} |\alpha_n^v(x)|$$

That is, $\mathrm{mag}(\alpha_n)$ is the absolute value of the largest output from any gate of α_n on any input $x \in [-2^n, 2^n]$.

DEFINITION 3.4. A family $\{\alpha_n\}$ of arithmetic circuits over R is *feasible-size-magnitude* if $\mathrm{size}(\alpha_n) = n^{O(1)}$ and $\mathrm{mag}(\alpha_n) = 2^{n^{O(1)}}$.

Again, these definitions can be extended in the obvious way to functions with many input and output arguments.

4. Arithmetic feasibility implies Boolean feasibility. Suppose that a function f can be computed by a feasible-size-magnitude family $\{\alpha_n\}$ of arithmetic circuits. Is f feasible in the sense that it can be computed by a polynomial size Boolean circuit?

How do we go from an arithmetic circuit that maps reals to reals to a Boolean circuit that maps bits to bits? One approach to establishing such a result is to use simulation. Suppose that we are given a description of a feasible-size-magnitude arithmetic circuit family $\{\alpha_n\}$ that sup-approximates some function f. We can try and construct a Boolean circuit family $\{\gamma_n\}$ such that γ_n exactly simulates the computation of α_n on certain special values of input x. Then the family $\{\gamma_n\}$ would compute f, and if the family were polynomial size, f would be feasible.

Assume that x is rational. Then we can simulate the computation of each α_n exactly using exact rational arithmetic. But α_n can have degree that is exponential in n, so even for small magnitude rational x, the number of bits required to exactly represent the output value of α_n grows exponentially. This results in exponential size for γ_n. So this direct simulation approach will not work.

However, $\{\alpha_n\}$ only approximates f, so we do not need to do an exact simulation. Instead we can approximately simulate each circuit on an approximations to the input x. The Boolean circuits doing the simulation will be polynomial size, provided that we do not need more than a polynomial number of bits of precision in the simulation. Thus we have the following theorem.

THEOREM 4.1. *If a real function f can be sup-approximated by a uniform family of feasible-size-magnitude arithmetic circuits then f is a feasible real function.*

Proof. Let $\{\alpha_n\}$ be a feasible-size-magnitude uniform arithmetic circuit family that sup-approximates f. Note that any approximation $\langle \alpha_n(x) \rangle_k$ to $\alpha_n(x)$ is going to be an approximation to $f(x)$ over some interval and with some degree of accuracy.

Since the Boolean circuit γ_n that computes approximation $\langle f(x) \rangle_n$ to $f(x)$ will actually be an approximation to an approximation of f, it will not be simulating α_n, but instead will simulate α_{n+1} over the interval $[-2^n, 2^n]$ by computing a $(n+1)$th approximation $\langle \alpha_{n+1}(x) \rangle_{n+1}$. That this is sufficient is because

$$\begin{aligned} |f(x) - \langle \alpha_{n+1}(x) \rangle_{n+1}| &\leq |f(x) - \alpha_{n+1}(x)| + |\alpha_{n+1}(x) - \langle \alpha_{n+1}(x) \rangle_{n+1}| \\ &\leq 2^{-n-1} + 2^{-n-1} \leq 2^{-n} \end{aligned}$$

Boolean circuit γ_n will simulate α_{n+1} on rational input x. (Recall that $\langle f(x)_n \rangle$ is by definition a rational number.) Since x is a rational, we could use a brute-force approach and just compute the value of $\alpha_{n+1}(x)$ exactly, by computing the value of each of the gates of α_{n+1}. The problem is that even though the output values do

not grow too large, the total number of bits required to represent the values exactly can grow exponentially with n. This would make the size of γ_n nonpolynomial and thus not feasible. To keep the simulation feasible, we must truncate the intermediate results corresponding to gates of α_{n+1}.

Each truncation causes a loss of precision that depends on the type of operation and magnitude of the values involved. Since the circuit family is feasible-size-magnitude, there is a function $m(n) = n^{O(1)}$ such that for $x \in [-2^n, 2^n]$ the output value of each gate in α_n has magnitude $\leq 2^{m(n)}$. This, combined with the results of Proposition 1.2 limit the loss of precision to at most $3m(n)$ bits per operation. Since there are at most $\text{depth}(\alpha_n)$ gates along the path from the output gate to x, at each gate at most $3m(n)$ more bits of precision are required of the inputs. So to compute the output of α_n within error 2^{-k} we need to have x within error $2^{-k-3m(n)\,\text{depth}(\alpha_n)}$. In other words, to compute an approximation $\langle \alpha_n(x) \rangle_k$ we need an approximation to x of $\langle x \rangle_{k+p(n)}$, where $p(n) \equiv 3m(n)\,\text{depth}(\alpha_n)$. Note that $p(n) = n^{O(1)}$.

Finally we can specify the Boolean circuit family $\{\gamma_n\}$ that feasibly computes $f(x)$. Remember that γ_n is computing $\langle \alpha_{n+1} \rangle_{n+1}$. The modulus function for the family is $\mu(n) \equiv p(n+1) + (n+1)$, and the range function is $\rho(n) \equiv m(n+1)$. Each γ_n takes as input the bits of $\langle x \rangle_{\mu(n)}$ and simulates α_{n+1} gate-by-gate using fixed-point binary arithmetic of range $\rho(n)$ and length $\rho(n) + \mu(n)$.

Since $\rho(n)$, $\mu(n)$, and $\text{size}(\alpha_{n+1})$ are all $n^{O(1)}$, the size of γ_n is also $n^{O(1)}$, and so the family $\{\gamma_n\}$ is feasible. Thus the function f is feasible as required. □

OPEN PROBLEM 4.2. *We must place a magnitude bound on the gate values in order to simulate the circuit family in feasible time. If we just place a limit on the magnitude of the output of the circuit and let the internal gates have arbitrarily large values can we compute a larger class of functions?*

5. Computing approximate binary representations. Before embarking on the proof of the converse of Theorem 4.1 we will examine a situation in which arithmetic circuits cannot possibly perform the same computations as Boolean ones. The problem will be to take a real number $x \in [-2^n, 2^n]$ and compute the bits of a range n length k fixed-point binary representation of an $(k-n)$th approximation to x.

Although the partial solution to this problem is crucial to the next section, the first time reader is urged to skip to the end of this section and examine just the statement of Proposition 5.4.

Let x be a real such that $x \in [-2^n, 2^n]$. We wish to compute bits $b_0, \cdots, b_k$ such that

$$\left| x - (2b_0 - 1)2^n \sum_{i=1}^{k} 2^{-i} b_i \right| \leq 2^{n-k}$$

The usual method of computing these bits involves performing mod operations or making comparisons—neither of which is directly available to an arithmetic circuit as a primitive operation. But let us suppose anyway that we have a comparison function defined for all rational a by

$$C_a(x) \equiv \begin{cases} 1 & \text{if } x \geq a \\ 0 & \text{otherwise} \end{cases}$$

The following program computes a range n length k fixed-point binary $(k-n)$th approximation to x:

$b_0 \leftarrow C_0(x)$
$x \leftarrow (2b_0 - 1)x$
for $i : 1..k$
  $b_i \leftarrow C_{2^{n-i}}(x)$
  $x \leftarrow x - b_i 2^{n-i}$
end for

The b_i resulting from this program represent a number $\langle x \rangle_{k-n}$ that is an $(k-n)$th approximation to x.

It is clear that $C_a(x)$ is a discontinuous function and thus cannot be sup-approximated by an arithmetic circuit. But it can be sup-approximated over a restricted range by a feasible family of arithmetic circuits. So, rather than having an exact $C_a(x)$ let us assume one that makes errors and behaves as follows on $x \in [-2^n, 2^n]$:

$$C_a^{p,n}(x) \in \begin{cases} [1 - 2^{-p}, 1] & \text{if } x \in [a + 2^{-p}, 2^n] \\ [0, 2^{-p}] & \text{if } x \in [-2^n, a - 2^{-p}] \\ [0, 1] & \text{if } x \in (a - 2^{-p}, a + 2^{-p}) \end{cases}$$

We call such a function a (p, s) *approximate comparison.*

Consider the same program as above but using approximate comparisons:

$\langle b_0 \rangle_p \leftarrow C_0^{p,n}(x)$
$x \leftarrow (2\langle b_0 \rangle_p - 1)x$
for $i : 1..k$
  $\langle b_i \rangle_p \leftarrow C_{2^{n-i}}^{p,n}(x)$
  $x \leftarrow x - \langle b_i \rangle_p 2^{n-i}$
end for

For any $x \in [-2^n, 2^n]$ this program produces some "approximate" bits $\langle b_i \rangle_p \in [0, 1]$. We would like each $\langle b_i \rangle_p$ to be a pth approximation to bit b_i of the fixed-point binary approximation we compute in the first program—thus our choice of notation. But this will only happen when each application of the approximate comparison function $C_a^{p,n}$ is to an argument outside of the error-prone interval $(a - 2^{-p}, a + 2^{-p})$. Under what circumstances will this happen?

Note that in the first program, where $C_a(x)$ works exactly, for each integer d, with $-2^k \leq d < 2^k$, the program computes the same binary representation for all $x \in (d2^{n-k}, (d+1)2^{n-k})$. That is, the interval $[-2^n, 2^n]$ is divided into 2^{k+1} intervals of width 2^{n-k}. In each interval, each real number is assigned the same binary representation. In the case of approximate comparisons, it is clear that $C_a^{p,n}(x)$ must not be permitted to make big errors, and so in order to keep the intermediate values of x away from the points prone to large errors x must initially be in the good interval $[d2^{n-k} + 2^{-p}, (d+1)2^{n-k} - 2^{-p}]$. At each step, when the comparison function makes small errors it has the same effect as a small shift in the original value of x. As long as x is not shifted out of its original interval, the $\langle b_i \rangle_p$ will continue to approximate a binary representation for the interval. (If it ever does leave the good interval then the resulting $\langle b_i \rangle_p$ will be a mixture of the first bits of one binary representation and the last bits of another, and so will not make sense.)

How much will x be shifted? Initially, when obtaining the sign bit $\langle b_0 \rangle_p$, x can be shifted by as much as

$$|(2b_0 - 1)x - (2\langle b_0 \rangle_p - 1)x| \leq 2|x||b_0 - \langle b_0 \rangle_p| \leq 2^{n+1-p}$$

At each iteration, the shift introduced is $\leq 2^{n-i-p}$ so the total shift for the entire program is $\leq 2^{n+1-p} + \sum_{i=1}^{k} 2^{n-i-p} \leq 2^{n+2-p}$.

So to ensure that x is not shifted out of the good interval it began in, it is sufficient that x initially be in the interval

$$[d2^{n-k} + 2^{-p} + 2^{n+2-p}, (d+1)2^{n-k} - 2^{-p} - 2^{n+2-p}]$$

The length of this interval is $2^{n-k} - 2^{-p+1} - 2^{n+3-p}$ and p must be chosen so that the length is positive. Choosing $p \geq k+5$ will work. We can use an even smaller interval length, and this gives us the following lemma.

LEMMA 5.1. *Let $x \in [-2^n, 2^n]$. For $p \geq k+5$, if for some integer d, $-2^k \leq d < 2^k$*

$$x \in [d2^{n-k} + 2^{n+3-p}, (d+1)2^{n-k} - 2^{n+3-p}]$$

then the $\langle b_0 \rangle_p, \cdots, \langle b_k \rangle_p$ produced by the preceding program are pth approximations to the bits $b_0, \cdots, b_k$ of a range n length k fixed-point binary representation of an $(k-n)$th approximation to x.

The next question is whether there is an arithmetic circuit family that computes $C_a^{p,n}$. This family must be feasible in p, n, and the length of a. The obvious approach of using polynomial approximations will not work, for the required degree is exponential in p and n. Fortunately, Newman [Ne64] shows how to approximate $|x|$ to within $3e^{-\sqrt{n}}$ using a degree n rational function. Using this result we can construct a feasible-size-magnitude arithmetic circuit family $\{\nu_n\}$ that behaves like $|x|$. This is then used to obtain an approximate comparison function.

LEMMA 5.2. *There is a feasible-size-magnitude arithmetic circuit family $\{\nu_n\}$ such that if $x \in [-1, 1]$ then*

$$| \, |x| - \nu_n(x) \, | \leq 2^{-n}$$

Proof. See Appendix A. □

Using the function ν we can construct another one that gives us the sign of its argument, with some error. Define the approximate sign function $S_p(x)$, for $p \geq 0$, by

$$S_p(x) \equiv \frac{x}{\nu_{2p+2}(x) + 2^{-2p-1}}$$

LEMMA 5.3. *The function $S_p(x)$ has the following property:*

$$S_p(x) \in \begin{cases} [1 - 2^{-p}, 1] & \text{if } x \in [2^{-p}, 1] \\ [-1, -1 + 2^{-p}] & \text{if } x \in [-1, -2^{-p}] \\ [-1, 1] & \text{if } x \in [-2^{-p}, 2^{-p}] \end{cases}$$

Proof. First consider the case of $x \in [-2^{-p}, 2^{-p}]$. So

$$\nu_{2p+2}(x) + 2^{-2p-1} \geq |x| - 2^{-2p-2} + 2^{-2p-1} \geq |x| + 2^{-2p-2}$$

and this gives $S_p(x) \in [-1, 1]$.

Now suppose $|x| \in [2^{-p}, 1]$. Since

$$0 < \nu_{2p+2}(x) + 2^{-2p-1} \leq |x| + 2^{-2p-2} + 2^{-2p-1} \leq |x| + 2^{-2p}$$

we have

$$\begin{aligned} |S_p(x)| &= \frac{|x|}{\nu_{2p+2}(x) + 2^{-2p-1}} \geq \frac{|x|}{|x| + 2^{-2p}} = \frac{1}{1 + |x^{-1}|2^{-2p}} \\ &\geq \frac{1}{1 + 2^p 2^{-2p}} \geq \frac{1}{1 + 2^{-p}} \geq 1 - 2^{-p} \end{aligned}$$

and so $|S_p(x)| \geq 1 - 2^{-p}$. Observing that $S_p(x)$ has the same sign as x establishes the remaining properties of S_p. □

We can now define the (p, n) comparison functions in terms of S_p by:

$$C_a^{p,n}(x) \equiv \frac{1 + S_{p+n}(2^{-n}(x - a))}{2}$$

It is easy to verify that it has the required properties, and that the circuits for producing the $\langle b_i \rangle_p$ are feasible-size-magnitude in terms of p, n, and the length of a fixed-point binary representation of a.

Thus we have the following proposition.

PROPOSITION 5.4. *There exists a feasible-size-magnitude family of arithmetic circuits $\{\beta_{n,k,p}\}$ with input x and outputs $\langle b_0 \rangle_p, \cdots, \langle b_k \rangle_p$ such that when $x \in [-2^n, 2^n]$, $k \geq n$, and $p \geq k + 5$ then:*

1. *Each $\langle b_i \rangle_p \in [0, 1]$.*
2. *If for some integer d, $-2^k \leq d < 2^k$, it is the case that*

$$x \in [d2^{n-k} + 2^{n+3-p}, (d+1)2^{n-k} - 2^{n+3-p}]$$

then the $\langle b_0 \rangle_p, \cdots, \langle b_k \rangle_p$ are pth approximations to the bits $b_0, \cdots, b_k$ of a range n length k fixed-point binary representation of an $(k-n)$th approximation to x given by

$$\langle x \rangle_{k-n} \equiv (2b_0 - 1)2^n \sum_{i=1}^{k} 2^{-i} b_i$$

3. *The b_i and $\langle b_i \rangle_p$ satisfy*

$$\left| \langle x \rangle_{k-n} - (2\langle b_0 \rangle_p - 1)2^n \sum_{i=1}^{k} 2^{-i} \langle b_i \rangle_p \right| \leq 2^{n+2-p}$$

Proof. For case 3. We have two cases to consider, $b_0 = 1$ and $b_0 = 0$. Suppose $b_0 = 1$. Then $\langle b_0 \rangle_p \in [1 - 2^{-p}, 1]$ and so $(2\langle b_0 \rangle_p - 1) \in [1 - 2^{-p+1}, 1]$. So there is some $\delta \in [0, 2^{-p+1}]$ such that

$$\begin{aligned} &\left| \langle x \rangle_{k-n} - (2\langle b_0 \rangle_p - 1)2^n \textstyle\sum_{i=1}^{k} 2^{-i} \langle b_i \rangle_p \right| \\ &= \left| \langle x \rangle_{k-n} - (1 - \delta)2^n \textstyle\sum_{i=1}^{k} 2^{-i} \langle b_i \rangle_p \right| \\ &\leq 2^n \textstyle\sum_{i=1}^{k} 2^{-i} \left| b_i - \langle b_i \rangle_p + \delta \right| \leq 2^n \textstyle\sum_{i=1}^{k} 2^{-i} \left(2^{-p} + 2^{-p+1}\right) \leq 2^{n+2-p} \end{aligned}$$

The case for $b_0 = 0$ is similar. □

6. Boolean feasibility implies arithmetic feasibility. Suppose f is a feasible real function computed by a Boolean circuit family $\{\gamma_n\}$. Since every continuous function f can be approximated by rational functions, there must be some arithmetic circuit family $\{\alpha_n\}$ that sup-approximates f. But is this family feasible-size-magnitude? The problem is that the feasibility of f is given by the discrete Boolean

computations that $\{\gamma_n\}$ performs to approximate f. Since we know almost nothing else about f, to get a feasible family of arithmetic circuits sup-approximating f we must refer to the original Boolean circuit. Taking advantage of the fact that f is continuous gives us the following theorem.

THEOREM 6.1. *If a real function f is feasible then f can be sup-approximated by a uniform family of feasible-size-magnitude arithmetic circuits.*

The proof of Theorem 6.1 is rather long, and so will be presented in stages. We begin with the well-known observation that we can simulate a Boolean circuit γ by an arithmetic circuit α with the same order of size and depth as γ. Suppose that we have available, as real numbers, the bits $b_0, \cdots, b_k$ that are input to a circuit γ. How can we compute γ's output bits $c_0, \cdots, c_l$ by an arithmetic circuit?

LEMMA 6.2. *Let γ be a Boolean circuit with inputs $b_0, \cdots, b_k$ and outputs $c_0, \cdots, c_l$. Then there exists an arithmetic circuit α over R, without $\cdot^{-1}$ gates, with inputs $\hat{b}_0, \cdots, \hat{b}_k$ and outputs $\hat{c}_0, \cdots, \hat{c}_l$ such that when the inputs $\hat{b}_i$ are restricted to be 0 or 1 the circuit α computes the same Boolean function as the circuit γ. Furthermore,* $\text{size}(\alpha) = O(\text{size}(\gamma))$ *and* $\text{depth}(\alpha) = O(\text{depth}(\gamma))$.

Sketch of proof. Assume that γ is constructed using only NAND and 0, 1 gates. Construct α as follows:

1. Associate each input b_i and output c_i of γ with input $\hat{b}_i$ and output $\hat{c}_i$ of α.
2. Associate Boolean constant 0, 1 gates of γ with rational constant gates 0, 1 of α.
3. Associate a NAND gate of γ computing $\neg(u \wedge v)$ with gates of α computing $1 - u \times v$.

Without changing the size or depth by more than a constant factor, we can always translate a Boolean circuit into one that just uses NAND gates and the constants 0, 1. Thus this argument works for any Boolean circuit. □

Now suppose that we have an arithmetic circuit α that simulates Boolean circuit γ. Circuit α computes a continuous function, in fact a polynomial, of its inputs. Thus small errors in the value of the inputs should only create small errors in the values of the outputs. So we should be able to simulate γ even if the inputs to α are not exactly 0 or 1.

LEMMA 6.3. *Let α be the arithmetic circuit of Lemma 6.2. Suppose that the inputs $\hat{b}_i$ to α are restricted to lying in the interval $[0, 1]$. Then the output of every gate in α is also restricted to $[0, 1]$ and*

$$|\alpha^i(\hat{b}_0, \cdots, \hat{b}_k) - \alpha^i(\langle \hat{b}_0 \rangle_{n+\text{depth}(\alpha)}, \cdots, \langle \hat{b}_k \rangle_{n+\text{depth}(\alpha)})| \leq 2^{-n}$$

where $\alpha^i(\hat{b}_0, \cdots, \hat{b}_k)$ denotes the value of output $\hat{c}_i$ of α on $\hat{b}_0, \cdots, \hat{b}_k$. That is, to compute approximation $\langle \hat{c}_i \rangle_n$ it is sufficient to have available the approximations $\langle \hat{b}_j \rangle_{n+\text{depth}(\alpha)}$.

Sketch of proof. We have two inductions based on the depth of α. To show that the output of every gate is $\in [0, 1]$ observe that level d of α contains 0, 1 gates or a gate computing $1 - uv$ of some gates u, v from a previous level. Thus $u, v \in [0, 1]$ and so is w.

The second induction is backwards. At level d, to compute the approximate value $\langle w \rangle_n$ of gate w computing $1 - uv$ we need $\langle u \rangle_{n+1}$ and $\langle v \rangle_{n+1}$. Thus to compute $\langle \hat{c}_i \rangle_n$ we may need $\langle \hat{x}_j \rangle_{n+\text{depth}(\alpha)}$. □

For all the arguments that follow, let real function f be computed by Boolean circuit family $\{\gamma_n\}$ with modulus function μ and range function ρ. Circuit γ_n takes a range n length $n + \mu(n)$ fixed-point binary encoding of the rational $\langle x \rangle_{\mu(n)}$, with bits

$b_0, \cdots, b_{n+\mu(n)}$, and outputs a range $\rho(n)$ length $\rho(n)+n$ fixed-point binary encoding of $\langle f(x)\rangle_n$, with bits $c_0, \cdots, c_{\rho(n)+n}$.

The actual simulation of Boolean circuit γ_n by an arithmetic circuit α_n is easy, assuming that you can get the inputs. Our problem is that the Boolean circuit γ_n wants the bits $b_0, \cdots, b_{n+\mu(n)}$ and all that we have available as input to α_n is the real number x. Also, the output of the arithmetic circuit is supposed to be a single real number, not a set of bits $c_0, \cdots, c_{\rho(n)+n}$.

Let us examine what γ_n is actually doing. The input bits b_i represent a $\mu(n)$th approximation to input x given by

$$\langle x\rangle_{\mu(n)} \equiv (2b_0 - 1)2^n \sum_{i=1}^{n+\mu(n)} 2^{-i} b_i$$

while the output bits c_i represent an approximation to $f(x)$ given by

$$\langle f(x)\rangle_n \equiv (2c_0 - 1)2^{\rho(n)} \sum_{i=1}^{\rho(n)+n} 2^{-i} c_i$$

Note that the latter expression can be computed exactly by a feasible-size-magnitude arithmetic circuit.

Suppose that we had a means of taking any real number $x \in [-2^n, 2^n]$ and computing a range n length $n + \mu(n)$ fixed-point binary number that is a $\mu(n)$th approximation to x. Then we could use an arithmetic circuit to exactly compute γ_n on the input bits b_i representing $\langle x\rangle_{\mu(n)}$, and get output bits c_i representing $\langle f(x)\rangle_n$. These can then be decoded exactly into a real number that is an nth approximation to $f(x)$.

This is illustrated by the pseudoarithmetic circuit family $\{\phi_n\}$ of Fig. 6.1. Since it exactly simulates the process of computing an approximation to $f(x)$ using γ_n, for $x \in [-2^n, 2^n]$ circuit ϕ_n satisfies $|f(x) - \phi_n(x)| \leq 2^{-n}$. We use the term "pseudo" because this circuit cannot actually exist, yet it can be approximated.

This approach would work fine, were it not for the problem that the bits b_i are not continuous functions of x, and so cannot possibly be computed by any arithmetic circuit. Instead, we will emulate the hypothetical circuit family $\{\phi_n\}$ using, in part, the approximate binary conversion circuit of Proposition 5.4, and construct an arithmetic circuit family $\{\Delta_{n,q}\}$ that will approximate $\{\phi_n\}$ for special subsets of the interval $[-2^n, 2^n]$.

PROPOSITION 6.4. *Let feasible real function f be computed by Boolean circuit family $\{\gamma_n\}$ with modulus function μ and range function ρ. There exists a feasible-size-magnitude circuit family $\{\Delta_{n,q}\}$ such that, for all $x \in [-2^n, 2^n]$, if*

$$x \in [d2^{-\mu(n)} + 2^{-q}, (d+1)2^{-\mu(n)} - 2^{-q}]$$

for some integer d, $-2^{n+\mu(n)} \leq d < 2^{n+\mu(n)}$, then

$$| f(x) - \Delta_{n,q}(x) | \leq 2^{-n+1}$$

Proof. We model the circuit $\Delta_{n,q}$ after the idealized circuit ϕ_n of Fig. 6.1. Let $\beta_{\cdot,\cdot,\cdot}$ be the approximate binary conversion circuit of Proposition 5.4, and let α_n be the arithmetic circuit of Lemma 6.2 that simulates γ_n. Construct $\Delta_{n,q}$ as in Fig. 6.2. Next, we must specify the functions $s(n)$ and $r(m,q)$. We first choose $s(n)$ so that

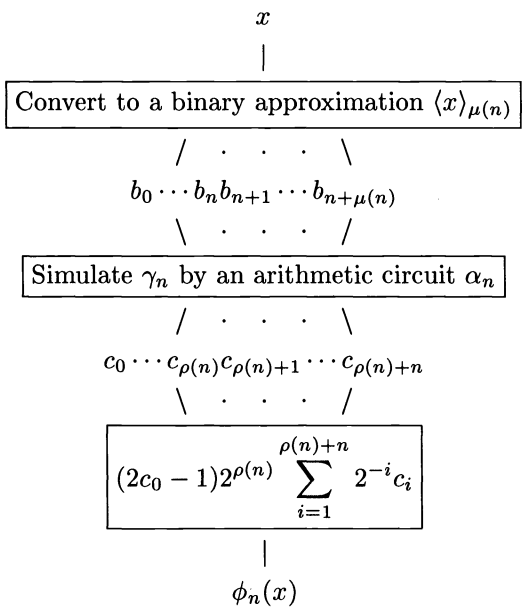


FIG. 6.1. *Pseudoarithmetic circuit ϕ_n approximating f.*

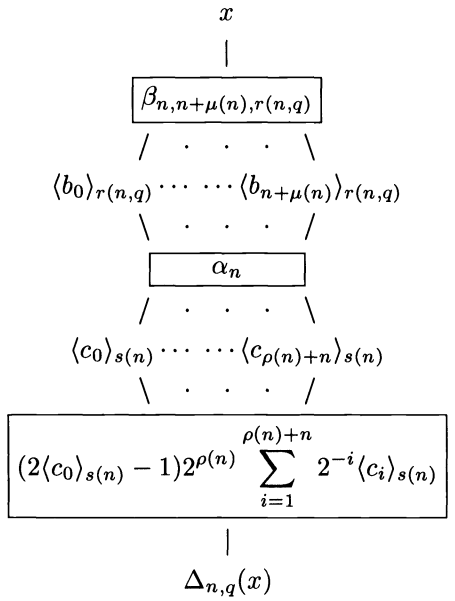


FIG. 6.2. *Arithmetic circuit $\Delta_{n,q}$ that almost approximates f.*

the error in the output of $\Delta_{n,q}$ introduced by approximating the c_i is $\leq 2^{-n}$. By Proposition 5.4 (3), we have that

$$\left| (2c_0 - 1)2^{\rho(n)} \sum_{i=1}^{\rho(n)+n} 2^{-i} c_i - (2\langle c_0 \rangle_{s(n)} - 1)2^{\rho(n)} \sum_{i=1}^{\rho(n)+n} 2^{-i} \langle c_i \rangle_{s(n)} \right| \leq 2^{\rho(n)+2-s(n)}$$

Defining $s(n) \equiv n + \rho(n) + 2$ is sufficient to ensure that the above quantity is $\leq 2^{-n}$.

The requirement that we need $s(n)$th approximations to the c_i places a minimum value on $r(n,q)$. By Lemma 6.3, to get the $\langle c_i \rangle_{s(n)}$ we need $\langle b_i \rangle_{r(n,q)}$ such that $r(n,q) \geq s(n) + \text{depth}(\alpha_n)$.

Applying Proposition 5.4 again, we have that circuit $\beta_{n,n+\mu(n),r(n,q)}$ delivers correct approximations to the b_i only when $r(n,q) \geq n + \mu(n) + 5$ and for those x that lie in the intervals

$$[d2^{-\mu(n)} + 2^{n+3-r(n,q)}, (d+1)2^{-\mu(n)} - 2^{n+3-r(n,q)}]$$

for integer d, with $-2^{n+\mu(n)} \leq d < 2^{n+\mu(n)}$.

We want $2^{n+3-r(n,q)} \leq 2^{-q}$ and so $r(n,q) \geq n + 3 + q$ is also required.

Combining these constraints on $r(n,q)$, we define

$$r(n,q) \equiv \max \begin{cases} s(n) + \text{depth}(\alpha_n) \\ n + \mu(n) + 5 \\ n + 3 + q \end{cases}$$

Note that $\text{depth}(\alpha_n) = O(\text{depth}(\gamma_n))$, $s(n) = n^{O(1)}$, and $r(n,q) = n^{O(1)}q^{O(1)}$. Thus the circuit $\Delta_{n,q}$ is of size $n^{O(1)}q^{O(1)}$, which is feasible.

Regardless of which value of $x \in [-2^n, 2^n]$ is input to $\Delta_{n,q}$, the approximations $\langle b_i \rangle_{r(n,q)}$ are always in $[0,1]$. Thus the $\langle c_i \rangle_{s(n)}$ are also restricted to $[0,1]$, and so $| \Delta_{n,q}(x) | \leq 2^{\rho(n)}$. Thus the circuit is also feasible-magnitude.

Finally, when x is in the interval

$$x \in [d2^{-\mu(n)} + 2^{-q}, (d+1)2^{-\mu(n)} - 2^{-q}]$$

the circuit $\Delta_{n,q}$ outputs an nth approximation to the output produced by γ_n on the input approximation $\langle x \rangle_{\mu(n)}$ given by the b_i. Since the output of γ_n is itself an nth approximation to $f(x)$ we have that $| f(x) - \Delta_{n,q}(x) | \leq 2^{-n+1}$. □

Now we have a problem. The circuit of Proposition 6.4 does not work for every x, as it makes errors on some sub-intervals of $[-2^n, 2^n]$, and so we cannot use it by itself to approximate f. Suppose that we had three different ways of computing $f(x)$ with the property that for any given x only one of the three made a significant error. Then we could take majority, in an approximate sense, and get the correct value of $f(x)$. The following three lemmas give a way of taking this kind of majority.

LEMMA 6.5. *Let x, y be arbitrary. Then*

$$\begin{aligned} \min\{x,y\} &= \frac{x + y - |x - y|}{2} \\ \max\{x,y\} &= \frac{x + y + |x - y|}{2} \end{aligned}$$

LEMMA 6.6. *Let z be arbitrary, and let x_i, for $i \in \{-1,0,1\}$ be such that for at least two values of i we have $|z - x_i| \leq 2^{-n}$. Let y be given by*

$$y \equiv x_{-1} + x_0 + x_1 - \min\{x_{-1}, x_0, x_1\} - \max\{x_{-1}, x_0, x_1\}$$

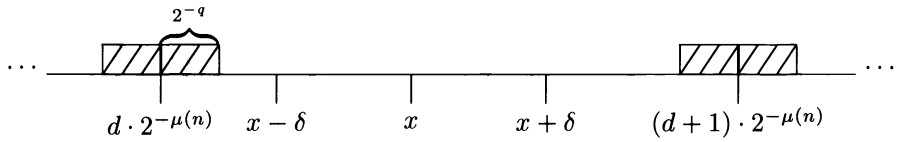
Then $|z - y| \leq 2^{-n}$.

LEMMA 6.7. *Let $z \in [-2^s, 2^s]$. Let $x_i \in [-2^s, 2^s]$, for $i \in \{-1,0,1\}$ be such that for at least two values of i, $|z - x_i| \leq 2^{-n-1}$. Then there is a feasible-size-magnitude arithmetic circuit family $\{\tau_{n,s}\}$ such that*

$$|z - \tau_{n,s}(x_{-1}, x_0, x_{+1})| \leq 2^{-n}$$

Proof. Use the approximation for $|x|$ given in Appendix A. □

We now return to the problem of approximating $f(x)$. By Lemma 2.5 we have that for $x, x' \in [-2^n, 2^n]$ if $|x - x'| \leq 2^{-\mu(n)}$ then $|f(x) - f(x')| \leq 2^{-n+1}$. Consider the following situation for an interval $[d2^{-\mu(n)}, (d+1)2^{-\mu(n)}]$. The shaded areas represent regions where the conversion to binary circuit makes big errors.



We wish to choose q and δ such that, for any value of x, at least two of x, $x - \delta$, $x + \delta$ will be outside the shaded area, and so will produce good approximations to $f(x)$.

LEMMA 6.8. *Let $q = \mu(n) + 4$ and $\delta = 2^{2-q}$. For any $x \in [-2^n, 2^n]$, at least two values of $i \in \{-1,0,1\}$ satisfy*

$$|f(x) - \Delta_{n,q}(x + i\delta)| \leq 2^{-n+2}$$

Now we can take majority and obtain the following proposition.

PROPOSITION 6.9. *Let feasible real function f be computed by Boolean circuit family $\{\gamma_n\}$ with modulus function μ and range function ρ. There exists a feasible-size-magnitude circuit family $\{\beta_n\}$ such that, for all $x \in [-2^n, 2^n]$, $| f(x) - \beta_n(x) | \leq 2^{-n}$.*

Proof. Define the family by

$$\beta_n(x) \equiv \tau_{n+1,1+\rho(n+2)}(\Delta_{n+2,q}(x - \delta), \Delta_{n+2,q}(x), \Delta_{n+2,q}(x + \delta))$$

where we choose $q = \mu(n+2) + 4$ and $\delta = 2^{2-q}$. Then τ_{n+1} is within 2^{-n-1} of $\Delta_{n+2,q}$ which is within 2^{-n-1} of f. Thus β_n is within 2^{-n} of f. □

This completes the proof of Theorem 6.1.

7. A feasible Weierstrass approximation theorem. Combining the results of Theorems 4.1 and 6.1 we get the following theorem.

THEOREM 7.1. *A real function f is feasible if and only if function f can be sup-approximated by a uniform family of feasible-size-magnitude arithmetic circuits.*

The Weierstrass approximation theorem states that every continuous function can, over a closed interval, be approximated arbitrarily closely by polynomials. We will show a feasible version of this theorem. That is, we will show that every feasible

real function can be approximated by a family of polynomials generated by feasible-size-magnitude circuits that do not contain any inverse gates.

Recall that Theorem 6.1 states that every feasible real function f can be sup-approximated by a uniform family $\{\alpha_n\}$ of feasible-size-magnitude arithmetic circuits. This circuit family uses inverse gates in a very restricted manner. In particular, only positive values ever appear as inputs to the gates. By replacing each inverse gate with a subcircuit that performs Newton's method we can approximate the inverse that would be computed by the original gate. We thus obtain an inverse-free feasible-size-magnitude circuit family $\{\beta_n\}$ that approximates $\{\alpha_n\}$ and thus approximates f. Since $\{\beta_n\}$ is inverse-free, each β_n is a polynomial, and we get a feasible version of the Weierstrass theorem.

THEOREM 7.2. *A real function f is feasible if and only if function f can be sup-approximated by a uniform family of inverse-free feasible-size-magnitude arithmetic circuits.*

Sketch of proof. Let f be computed by Boolean circuit family $\{\gamma_n\}$. Consider the circuit family $\{\alpha_n\}$ of Theorem 7.1 that was derived from $\{\gamma_n\}$. We have for $x \in [-2^n, 2^n]$ that $|f(x) - \alpha_n(x)| \leq 2^{-n}$.

Now consider Theorem 4.1 in which we simulate an arithmetic circuit family by a Boolean circuit family. This theorem is really just a backward error analysis of each circuit, and it specifies how much internal accuracy is needed by the Boolean circuit to simulate the arithmetic circuit to a given output accuracy. For feasible-size-magnitude circuits, to compute $\langle \alpha_n(x) \rangle_n$ we need only do all internal computations to precision $2^{-n^{O(1)}}$. In effect, this says that the arithmetic circuits can make small errors at each gate and still compute a good approximation to f. So inverses need not be computed exactly, and we can replace the gates by subcircuits that approximate the inverse using the Newton iteration of Appendix B.

In the construction of α_n all inverses are of positive values. Also, because the circuits are feasible-magnitude, the inputs to the inverse gate range between $2^{-n^{O(1)}}$ to $2^{n^{O(1)}}$ in magnitude. Thus the circuit for the Newton iteration needs only do $n^{O(1)}$ steps. It is feasible-size-magnitude.

As a result of the replacement we get a new inverse-free, feasible-size-magnitude circuit family $\{\beta_n\}$ with the property that $|\alpha_n(x) - \beta_n(x)| \leq 2^{-n}$ for $x \in [-2^n, 2^n]$. Thus $|f(x) - \beta_{n+1}(x)| \leq 2^{-n}$ and we have the theorem. □

One application of the general Weierstrass approximation theorem is to translate the integration of arbitrary functions to the polynomial domain. Integration then becomes a term-by-term operation on easy to manage polynomials. This occurs both in classical functional analysis and in recursive analysis. But in our version, each member of the circuit family $\{\beta_n\}$ is a very compact representation of a polynomial that may have degree greater than 2^n, may have many nonzero coefficients, and possibly have non-feasible-magnitude coefficients. Thus efficient integration requires a more clever approach than just term-by-term evaluation.

OPEN PROBLEM 7.3. *Precisely characterize the coefficients of the polynomials computed in Theorem 7.2. We conjecture that the coefficients of β_n are bounded by $2^{n^{O(1)}}$.*

In Problem 2 of Kaltofen [Ka87] it is asked if given a family of polynomials, $\{P_n\}$, computed by polynomial-length straight-line programs whether in general the anti-derivatives of the family can also be computed by polynomial-length straight-line programs. Suppose that we restrict the problem further, and require that the polynomials be generated by a feasible-size-magnitude circuit family $\{\alpha_n\}$. In addition, we

relax the condition that the anti-derivatives must be computed exactly. Is it possible to approximate the anti-derivatives with a feasible-size-magnitude circuit family? The following result indicates that this is unlikely.

THEOREM 7.4. *There is a family* $\{\alpha_n\}$ *of inverse-free feasible-size-magnitude arithmetic circuits such that the following are equivalent:*

1. *There is a family* $\{\beta_n\}$ *of feasible-size-magnitude arithmetic circuits such that, for* $x \in [-2^n, 2^n]$

$$\left| \beta_n(x) - \int_0^x \alpha_n(y)\mathrm{d}y \right| \leq 2^{-n}$$

2. $P = \#P$

In other words, it seems unlikely that every polynomial generated by a feasible-size-magnitude circuit has an anti-derivative that can be generated by a feasible-size-magnitude circuit. So, even though approximation is easier than exact computation, it is still not easy enough. The essence of this result (see [Fr84], [Ho87]) is that we can encode satisfiability by a feasible real function that has peaks at each satisfying assignment. Then we can use integration to detect and count peaks, and thus count satisfying assignments.

It should be remarked that these results extend to multiple argument functions in the obvious way, and also to functions and circuits over the complex numbers.

8. Other results. An important result in algebraic complexity theory is that divisions do not help much when computing polynomials of small degree (see [Str73], [BGH82], [Ka87]). That is, every circuit of size s and degree d with divisions can be transformed into an equivalent division-free circuit of size $s^{O(1)}d^{O(1)}$. As long as the size and degree of the circuits in a family are polynomial, one obtains an equivalent division-free family of polynomial size and depth. The basic idea of the transformation is due to Strassen. Each operation is replaced by its formal power series about some point. For example, replace $1/x$ with the power series $1/(1-(1-x)) = 1+(1-x)+(1-x)^2+(1-x)^3+\cdots$. Then all operations are carried out in the domain of formal power series. Since the circuit computes a polynomial, the power series can be truncated at the degree of the polynomial and a division-free circuit results.

Suppose we forget about computing real functions, and just look at uniform families of feasible-size-magnitude arithmetic circuits as a collection of different functions. That is, in family $\{\alpha_n\}$, circuits α_i and α_j may be computing rational functions that are very dissimilar in behaviour. Can we construct an inverse-free feasible-size-magnitude circuit family $\{\beta_{n,k}\}$ such that $\beta_{n,k}$ is a kth approximation to α_n?

We cannot use Strassen's technique because our circuits can have exponential degree and so the truncated power series does not have a polynomial number of terms. But our notion of computation does not require us to compute exactly, and so we need only compute approximations to $1/x$. We still cannot use the power series for $1/x$ this time because it does not converge fast enough to get a good approximation.

What we can do is replace inversions by Newton's method as we did for the feasible Weierstrass approximation theorem. There is one slight catch—we need to know the sign of the element we are inverting in order to get the correct starting value for the iteration.

A non-log-space uniform method of doing this transformation is as follows. By definition, no inverse gate of an arithmetic circuit is allowed to have an input of zero. Also, the output of every gate is a continuous function of x. Thus we know that the input to an inverse gate cannot change sign. If the circuit is feasible-magnitude, then

the input to the inverse gate cannot get any smaller in magnitude than $2^{-n^{O(1)}}$. This means we can simulate the values of the gates in α_n on input $x = 0$ and determine the sign of the input to each inverse gate. Then we know whether to use a positive or negative starting value for the iteration.

Although we can do this simulation in polynomial time, it is not clear how to do it directly in log space and so preserve uniformity. The circuit resulting from the polynomial time simulation is however P-uniform [BCH86].

To preserve log-space uniformity we must take a less direct route. The idea is to use the same techniques as Theorem 4.1 to simulate each α_n by a family of Boolean circuits $\{\gamma_{n,k}\}$ such that $\beta_{n,k}$ kth approximates α_n. Then, apply the techniques of Theorem 6.1 to simulate this Boolean family by an arithmetic family $\{\beta_{n,k}\}$ such that $\beta_{n,k}$ will kth approximate α_n. This new family has only positive inputs to its inverse gates. Finally, remove inverses from $\beta_{n,k}$ using Newton's method. All of the technical details required are reasonable modifications of the proofs of Theorems 4.1 and 6.1, mainly just adding the parameter k to the various internal functions.

Thus we get the following theorem.

THEOREM 8.1. *Let $\{\alpha_n\}$ be a uniform family of feasible-size-magnitude arithmetic circuits. Then there exists a uniform family $\{\beta_{n,k}\}$ of inverse-free feasible-size-magnitude arithmetic circuits that, for $x \in [-2^n, 2^n]$, have $|\alpha_n(x) - \beta_{n,k}(x)| \leq 2^{-k}$.*

OPEN PROBLEM 8.2. *Is there a more direct way of transforming α_n to β_n in log space?*

OPEN PROBLEM 8.3. *If we restrict the α_n to computing polynomials, can we remove inverses and still compute the original polynomial* exactly*?*

Finally, in all of the preceding theorems, we can replace feasible-size-magnitude by feasible-depth-magnitude (polynomial depth rather than size) to get analogous results for depth and space. These other results will appear in a subsequent paper.

Appendix A. Circuits based on Newman's theorem.

THEOREM A.1 (Newman). *For $n \geq 4$ let*

$$P_n(x) \equiv \prod_{k=0}^{n-1} (x + e^{-k/\sqrt{n}})$$

and

$$R_n(x) \equiv x \frac{P_n(x) - P_n(-x)}{P_n(x) + P_n(-x)}$$

Then for $x \in [-1, 1]$

$$| \, |x| - R_n(x) \, | \leq 3e^{-\sqrt{n}}$$

Proof. See [Ne64]. □

For fixed n, no arithmetic circuit can compute R_n since we cannot generate the constant $e^{-1/\sqrt{n}}$ exactly. But we should be able to use an approximation to this instead and still obtain a function that behaves like R_n.

One worry we have is that the denominator of R_n might get too small and make the output of the inverse gate too big, and thus the circuit would not have feasible-magnitude. Fortunately this does not happen.

LEMMA A.2. *For $x \in [-1, 1]$ and $n > 4$*

$$|P_n(x) + P_n(-x)| \geq 2^{-n\sqrt{n}-1}$$

Proof. For $x \in [0,1]$ we have

$$P_n(x) \geq \prod_{k=0}^{n-1} e^{-k/\sqrt{n}} = e^{-n(n-1)/2\sqrt{n}} \geq e^{-n\sqrt{n}/2}$$

In the proof in [Ne64] it is shown that for $e^{-\sqrt{n}} \leq x \leq 1$ we have $|P_n(-x)| \leq |P_n(x)|e^{-\sqrt{n}}$. Thus for $n > 1$

$$\begin{aligned} |P_n(x) + P_n(-x)| &\geq |P_n(x)| - |P_n(-x)| \geq |P_n(x)|(1 - e^{-\sqrt{n}}) \\ &\geq \frac{1}{2}|P_n(x)| \geq \frac{1}{2}e^{-n\sqrt{n}/2} \end{aligned}$$

For $-1 \leq x \leq -e^{-\sqrt{n}}$ a similar argument gives $|P_n(x) + P_n(-x)| \geq \frac{1}{2}e^{-n\sqrt{n}/2}$.

For $-e^{-\sqrt{n}} < x < e^{-\sqrt{n}}$ we have $P_n(x) > 0$ and so $|P_n(x) + P_n(-x)| \geq e^{-n\sqrt{n}/2}$. Thus $|P_n(x) + P_n(-x)| \geq \frac{1}{2}e^{-n\sqrt{n}/2} \geq 2^{-2(n\sqrt{n}/2)-1}$. □

The next thing to do is find a suitable approximation to P_n.

LEMMA A.3. *There is a feasible-size-magnitude arithmetic circuit family* $\{\nu'_{n,m}\}$ *such that for* $x \in [-1,1]$

$$|P_n(x) - \nu'_{n,m}(x)| \leq 2^{-m}$$

Proof. We wish to compute $\langle P_n(x)\rangle_m$ for $x \in [-1,1]$, where $P_n(x) \equiv \prod_{k=0}^{n-1}(x + e^{-k/\sqrt{n}})$. One way to do this is with a $N = \lceil \lg n \rceil$ depth binary tree of multiplications. The leaves of the tree are the terms $(x + e^{-k/\sqrt{n}})$. Each of the terms has magnitude less than or equal to 2 and so $|P_n(x)| \leq 2^{2^N}$. The numbers computed at level i of the tree, with the root level 0 and the leaves level $N-1$, have magnitude bounded by $2^{2^{N-i}}$. To get an mth approximation to the result at the root of the tree we apply Proposition 1.2 and see that we need at worst an lth approximation to the leaves, where

$$l \geq m + 2^{N-1} + 2 + 2^{N-2} + 2 + \cdots + 2^{N-N} + 2 = m + 2N + 2^N - 1$$

Thus to obtain $\langle P_n(x)\rangle_m$ it is sufficient to have approximations $\langle x + e^{-k/\sqrt{n}}\rangle_M$ for $0 \leq k < n$, where $M = m + 2N + 2^N - 1$. Note that $M = O(m+n)$. To get these, we just need to take powers of the approximation $\langle e^{-1/\sqrt{n}}\rangle_M$. This latter approximation can be computed by taking at most M terms of the power series for $e^{-1/\sqrt{n}}$, using approximation $\langle 1/\sqrt{n}\rangle_M$ instead of $1/\sqrt{n}$.

It is simple to verify that the resulting arithmetic circuit has a size of $n^{O(1)}m^{O(1)}$ and a depth of $O(\log nm)$, and that all of the intermediate gate values are feasible-magnitude. □

With this we can construct a circuit that behaves essentially like the function R_n.

LEMMA A.4. *There is a feasible-size-magnitude arithmetic circuit family* $\{\nu_n\}$ *such that if* $x \in [-1,1]$ *then*

$$|\,|x| - \nu_n(x)| \leq 2^{-n}$$

Proof. Sketch. The circuit ν_n will exactly compute the formula

$$x \times \left((\nu'_{N,M}(x) - \nu'_{N,M}(-x)) \times (\nu'_{N,M}(x) + \nu'_{N,M}(-x))^{-1}\right)$$

where M, N will be chosen so that $| \, |x| - \nu_n(x)| \leq 2^{-n}$ for $x \in [-1, 1]$. In effect, ν_n is an mth approximation to R_N for some m. Since $|P_N(x) + P_N(-x)| \geq 2^{-N\sqrt{N}-1}$ it is easy to see that to compute $\langle R_N(x) \rangle_m$ it is sufficient to have approximations $\langle P_N(x) \rangle_{m+4N\sqrt{N}+8}$. Thus one condition on M is that $M \geq m + 4N\sqrt{N} + 8$.

Now

$$\begin{aligned} | \, |x| - \nu_n(x) \, | &\leq | \, |x| - R_N(x)| + |R_N(x) - \nu_n(x)| \\ &\leq 3e^{-\sqrt{N}} + 2^{-m} \leq 2^{2-\sqrt{N}} + 2^{-m} \end{aligned}$$

and we want this to be $\leq 2^{-n}$, that is, $m \geq n + 1$ and $\sqrt{N} - 2 \geq n + 1$. We can achieve this by setting $N = (n + 3)^2$ and $M = m + 4N\sqrt{N} + 8 = n + 9 + 4(n + 3)^3$.

Now ν_n is certainly feasible-size. We also must ensure that it is feasible-magnitude. The only potential problem is the inverse gate. Since $|P_N(x) + P_N(-x)| \geq 2^{-N\sqrt{N}-1}$ as long as $M \geq N\sqrt{N}+3$, then $|\nu'_{N,M}(x)+\nu'_{N,M}(-x)| \geq 2^{-N\sqrt{N}-2}$ and so the inverse gate has magnitude $\leq 2^{N\sqrt{N}+2}$, which is feasible. □

Appendix B. Newton's method for computing inverses. The following proposition gives a division-free iteration for computing x^{-1} using Newton's method.

PROPOSITION B.1. *Let x be a real such that $2^{-k} < x < 2^k$. Let the sequence $\{y_i\}$ be defined by*

$$\begin{aligned} y_0 &= 2^{-k} \\ y_{i+1} &= y_i(2 - xy_i) \quad \text{for } i \geq 0 \end{aligned}$$

Then $|x^{-1} - y_i| \leq 2^{-n}$ for $i \geq 3k + \lg(k + n)$.

Proof. Recall Newton's method for finding a zero of a real function f via the iteration

$$y_{i+1} = y_i - \frac{f(y_i)}{f'(y_i)}$$

If we define $f(y) \equiv x^2 - x/y$ then $f(x^{-1}) = 0$. Since $f'(y) = x/y^2$ we get the iteration

$$y_{i+1} = y_i - \left(x^2 - \frac{x}{y_i}\right)\frac{y_i^2}{x} = 2y_i - xy_i^2 = y_i(2 - xy_i)$$

Define the error δ_i in y_i by $\delta_i \equiv x^{-1} - y_i$.

The iteration has two fixed points, $y = 0$ and $y = x^{-1}$. In order for the iteration to converge to x^{-1} it is necessary that $(2 - xy_i) > 0$. Note that if y_i ever becomes negative or zero it will remain so. The next two lemmas show that the iteration will converge to x^{-1} from below if y_0 begins at a value $< x^{-1}$.

LEMMA B.2. *If $0 < xy_0 < 1/2$ then there is an $l > 0$ such that the sequence $\{y_i\}$ has the property that*

$$\frac{3}{2}y_i < y_{i+1} \quad \text{for } 0 \leq i \leq l$$

and

$$\begin{aligned} 0 < xy_i < \frac{1}{2} \quad &\text{for } 0 \leq i < l \\ \frac{1}{2} \leq xy_l < 1 \end{aligned}$$

Proof. Note that $y_{i+1} = y_i(2 - xy_i)$ and so $\frac{3}{2}y_i < y_{i+1} < 2y_i$. Thus the y_i form a monotone increasing sequence. Let l be the index of the first term y_l in the sequence for which $xy_l \geq 1/2$. Since $0 < xy_{l-1} < 1/2$ we have $xy_l < x2y_{l-1} < 1$ and so $1/2 \leq xy_l < 1$. □

LEMMA B.3. *Let l be such that $1/2 \leq xy_l < 1$. Then the sequence of approximations $\{y_l, y_{l+1}, \cdots\}$ has the property that*

$$\frac{1}{2} \leq xy_{l+i} < 1 \quad \text{for } i \geq 0$$

and

$$\delta_{l+i} \leq x^{-1} \cdot 2^{-2^i}$$

Proof. First note that

$$\begin{aligned} y_{i+1} &= y_i(2 - xy_i) = (x^{-1} - \delta_i)(2 - x(x^{-1} - \delta_i)) \\ &= (x^{-1} - \delta_i)(1 + x\delta_i) = x^{-1} - x\delta_i^2 \end{aligned}$$

and so $\delta_{i+1} = x\delta_i^2$.

If $1/2 \leq xy_i < 1$ then $\frac{1}{2}x^{-1} \leq y_i < x^{-1}$ and so $0 < \delta_i \leq \frac{1}{2}x^{-1}$. Thus $\frac{3}{4}x^{-1} < y_{i+1} = x^{-1} - x\delta_i^2 < x^{-1}$, which means that $3/4 < xy_{i+1} < 1$.

Since $\delta_{i+1} = x\delta_i^2$ a simple induction shows that $\delta_{l+i} = x^{2^i-1}\delta_l^{2^i}$. Now $0 < \delta_l \leq \frac{1}{2}x^{-1}$ so we get $\delta_{l+i} \leq x^{2^i-1}x^{-2^i} \cdot 2^{-2^i} = x^{-1} \cdot 2^{-2^i}$. □

Returning to the proof of the proposition, assume that $2^{-k} < x < 2^k$. By choosing $y_0 = 2^{-k}$ then either Lemma B.2 or B.3 applies. The worst case is when Lemma B.2 applies followed by Lemma B.3. When B.2 applies we have that

$$y_i \geq 2^{-k}(\frac{3}{2})^i$$

An upper bound l on the first i for which $xy_i \geq 1/2$ is given by the minimum l for which

$$xy_l \geq 2^{-k}y_l \geq 2^{-k}2^{-k}(\frac{3}{2})^l \geq \frac{1}{2}$$

That is $-2k + l(\lg 3 - 1) \geq -1$ or $l \geq (2k-1)/(\lg 3 - 1)$, which gives an upper bound of $l < 3k$. So at most $3k$ steps of the iteration are required before Lemma B.3 applies.

When it does, the sequence converges quadratically with $\delta_{l+i} \leq x^{-1}2^{-2^i}$. Thus to get $\delta_{l+i} \leq 2^{-n}$ requires at most i steps where i is the minimum such that $2^k2^{-2^i} \leq 2^{-n}$. That is $2^i \geq k + n$ or at most $\lg(k+n)$ more steps of the iteration are required.

So $|x^{-1} - y_i| \leq 2^{-n}$ for $i \geq 3k + \lg(k+n)$ and we have the proposition. □

REFERENCES

[BCH86] 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.

[Bo77] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[BGH82] A. BORODIN, J. VON ZUR GATHEN, AND J. HOPCROFT, *Fast parallel matrix and* gcd *computations*, Inform. and Control, 52 (1982), pp. 241–256.

[BB88] J. M. BORWEIN AND P. B. BORWEIN, *On the complexity of familiar functions and numbers*, SIAM Rev., 30 (1988), pp. 589–601,

[Fr84] H. FRIEDMAN, *The computational complexity of maximization and integration*, Adv. in Math., 53 (1984), pp. 80–98.

[vzG87] J. VON ZUR GATHEN, *Feasible arithmetic computations: Valiant's hypothesis*, J. Symb. Comput., 4 (1987), pp. 137–172.

[vzGS86] J. VON ZUR GATHEN AND G. SEROUSSI, *Boolean circuits versus arithmetic circuits*, in Proc. 6th Internat. Conference in Computer Science, Santiago, Chile, 1986, pp. 171–184.

[Gr57] A. GRZEGORCZYK, *On the definition of computable real continuous functions*, Fund. Math., 44 (1957), pp. 61–71.

[Ho87] H. J. HOOVER, *Feasibly constructive analysis*, Ph.D. thesis and Tech. Report 206/87, Department of Computer Science, University of Toronto, Toronto, Canada, 1987.

[Ka87] E. KALTOFEN, *Single-factor Hensel lifting and its application to the straight-line complexity of certain polynomials*, in Proc. 19th ACM Symposium on Theory of Computing, Association for Computing Machinery, 1987, pp. 443–452.

[KF82] K. KO AND H. FRIEDMAN, *Computational complexity of real functions*, Theoret. Comput. Sci., 20 (1982), pp. 323–352.

[Ne64] D. J. NEWMAN, *Rational approximation to* $|x|$, Michigan Math. J., 11 (1964), pp. 11–14.

[Sh76] J. C. SHEPHERDSON, *On the definition of computable function of a real variable*, Z. Math. Logik Grundlag. Math., 22 (1976), pp. 391–402.

[Str73] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 182–202.

[Tu36] A. M. TURING, *On computable numbers, with an application to the entscheidungsproblem*, Proc. London Math. Soc., (2), 42 (1936/37), pp. 230–265.

[Tu37] ———, *A correction*, Proc. London Math. Soc., (2), 43 (1937), pp. 544–546.

[Va82] L. G. VALIANT, *Reducibility by algebraic projections*, in Logic and Algorithms, Symposium in honor of Ernst Specker, L'Enseignment Mathématique, 30 (1982), pp. 365–380.

SIAM J. COMPUT.
Vol. 19, No. 1, pp. 205–206, February 1990

013

ERRATUM: Generalized Selection and Ranking: Sorted Matrices*

GREG N. FREDERICKSON† AND DONALD B. JOHNSON‡

The proof of Lemma 7 contains several arithmetic mistakes, which can be corrected easily while still maintaining the basic form of the proof [*SIAM J. Comput.*, 13(1984), pp. 14–30]. We give a revised proof for this lemma.

LEMMA 7. *Selection of a kth element in a sorted matrix X of dimension $n \times m$, $1 < m \leq n$, requires time $\Omega(h \log(2k/h^2))$, where $h = \min\{\sqrt{k}, m\}$ and $k \leq \lceil nm/2 \rceil$.*

Proof. If $k < 13$, then clearly the lemma holds. Thus we consider the case in which $k \geq 13$. We construct a basic "step-shaped" configuration as follows. If $m = 2$, an initial configuration will have k elements in column 1. A valid configuration may be obtained by moving up to $\lfloor k/2 \rfloor$ elements into column 2. If $m = 3$, an initial configuration will have $\lfloor 2k/3 \rfloor$ elements in column 1 and $\lceil k/3 \rceil$ elements in column 2. This allows moving up to $\lfloor k/3 \rfloor$ elements from column 1 to column 3. If $m = 4$, an initial configuration will have $\lceil k/2 \rceil$ elements in column 1 and $\lfloor k/2 \rfloor$ elements in column 2. This allows moving up to $\lfloor k/4 \rfloor$ elements from column 2 to column 3. So, for the cases $m = 2$, $m = 3$, and $m = 4$, $P = \Omega(k)$ and $\log P = \Omega(\log k)$, which is $\Omega(h \log(2k/h^2))$.

For $m \geq 5$, we have the following. Let $a = \min\{m, \lfloor \sqrt{2k} \rfloor\}$, $s = \lfloor 2k/a^2 \rfloor$, and $b = \lceil k/a + (s/a)\lfloor a/2 \rfloor (a + \lceil a/2 \rceil + 1)/2 \rceil$. Note that $s \geq 1$. The basic configuration has $b - s(i-1)$ elements in each column $i = 1, \cdots, \lfloor a/2 \rfloor$, and a total of $k - \sum_{i=1}^{\lfloor a/2 \rfloor} (b - s(i-1))$ elements in columns $i = \lfloor a/2 \rfloor + 1, \cdots, a$, with each of the latter columns containing no more than $b - s\lfloor a/2 \rfloor$ elements, and no more elements in column i than in column $i - 1$. It is possible to move up to s elements from any column $i = 1, \cdots, \lfloor a/2 \rfloor$ to columns with index greater than $\lfloor a/2 \rfloor$, while still observing the restrictions on each of the latter columns, and this may be done independently for each of the first $\lfloor a/2 \rfloor$ columns. Therefore $P \geq (s+1)^{\lfloor a/2 \rfloor}$ and $\log P = \Omega(h \log(2k/h^2))$ as required.

We now verify that such configurations exist for all values of k, m, and n for which $13 \leq k \leq \lceil nm/2 \rceil$ and $n \geq m \geq 5$. It may be verified that $s\lfloor a/2 \rfloor \leq b \leq n$. From the definition of b, we have

$$b \geq k/a + (s/a)\lfloor a/2 \rfloor (a + \lceil a/2 \rceil + 1)/2$$

which is equivalent to

$$ab - s\lfloor a/2 \rfloor (a + \lceil a/2 \rceil + 1)/2 \geq k$$

which is equivalent to

$$b\lfloor a/2 \rfloor - s\lfloor a/2 \rfloor (\lfloor a/2 \rfloor + 1)/2 + \lceil a/2 \rceil (b - s\lfloor a/2 \rfloor) \geq k$$

which is equivalent to

$$\sum_{i=1}^{\lfloor a/2 \rfloor} (b - si) + \sum_{i=\lfloor a/2 \rfloor + 1}^{a} (b - s\lfloor a/2 \rfloor) \geq k.$$

* Received by the editors October 10, 1989; accepted for publication October 16, 1989.
† Department of Computer Sciences, Purdue University, West Lafayette, Indiana 47907.
‡ Department of Computer Science, Dartmouth College, Hanover, New Hampshire 03755.

From the above inequality it is clear that there is a valid configuration with the minimum number of elements, $b - si$, in each column $i = 1, \cdots, \lfloor a/2 \rfloor$.

We next show the existence of a basic configuration, i.e., one that has $b - s(i-1)$ elements in columns $i = 1, \cdots, \lfloor a/2 \rfloor$. Since $m \geq 5$ and $k \geq 13$, we have $a \geq 5$, which implies that

$$8k \leq 3ak - 6k$$

Since $a \leq \sqrt{2k}$, this implies

$$4a^2 \leq 3ak - 6k$$

which is equivalent to

$$(2k/a^2)(a+6)/8 + 1 \leq k/a$$

which, using the definition of s, implies

$$s(a+6)/8 + 1 \leq k/a$$

which is equivalent to

$$k/a + (s/a)(a^2/8 + a/4) + 1 \leq 2k/a - s/2$$

which implies

$$k/a + (s/a)\lfloor a/2 \rfloor \lceil a/2 \rceil /2 + (s/a)\lfloor a/2 \rfloor /2 + 1 \leq k/\lfloor a/2 \rfloor - s/2$$

which is equivalent to

$$k/a + (s/a)\lfloor a/2 \rfloor (a + \lceil a/2 \rceil + 1)/2 + 1 \leq k/\lfloor a/2 \rfloor + s(\lfloor a/2 \rfloor - 1)/2$$

which, using the definition of b, implies

$$b \leq k/\lfloor a/2 \rfloor + s(\lfloor a/2 \rfloor - 1)/2$$

which is equivalent to

$$b\lfloor a/2 \rfloor - s\lfloor a/2 \rfloor (\lfloor a/2 \rfloor - 1)/2 \leq k$$

which is equivalent to

$$\sum_{i=1}^{\lfloor a/2 \rfloor} (b - s(i-1)) \leq k.$$

Thus the required configurations always exist. □

REFERENCES

[1] G. N. FREDERICKSON AND D. B. JOHNSON, *Generalized selection and ranking: Sorted matrices*, SIAM J. Comput., 13 (1984), pp. 14–30.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 207-221, April 1990

REVERSAL COMPLEXITY CLASSES FOR ALTERNATING TURING MACHINES*

MIROSŁAW KUTYŁOWSKI†, MACIEJ LIŚKIEWICZ†, AND KRZYSZTOF LORYŚ†

Abstract. Alternating Turing machines (ATMs) with bounded number of reversals are considered. It is proved that the machines making fewer than $\log^* n$ reversals can recognize only regular languages. On the other hand, the class of languages that can be recognized by ATMs using $\log^* n$ reversals is very wide. The authors prove that above this limit even a slight increase of the number of reversals leads to a considerably larger class of languages. It is also proved that every $T(n)$-time bounded ATM may be replaced by an equivalent machine working in the same time and making no more than $\log^*(T(n))$ reversals.

Key words. alternation, reversal complexity, regular languages, computational complexity

AMS(MOS) subject classifications. 68Q05, 68Q15

1. Introduction. Alternating Turing machines (ATMs) were introduced by Chandra, Kozen, and Stockmeyer [4], and since then ATMs have been intensively studied. In particular, many results on alternating time and space complexity have been proved.

Along with the time and space complexity, the reversal complexity is an important and widely used complexity measure. It was introduced by Hartmanis [11] and Fischer [8], and then it was investigated for many classes of problems and different types of computational devices, especially for pushdown and counter automata and for one-tape Turing machines. Gurari and Ibarra [9] proved that deterministic multicounter machines that make a constant number of reversals on counters cannot recognize any nonregular unary language. It is not true for nondeterministic multipushdown machines (Baker and Book [2]) or even for deterministic ones (Chrobak [5]). One-tape ATMs with a constant number of reversals were considered by Liśkiewicz, Loryś, and Piotrów [13]. They showed that such machines recognize only regular languages.

It is known that two-tape nondeterministic Turing machines making only one reversal recognize all recursively enumerable languages (see, e.g., Baker and Book [2]). So it is reasonable to assume that all considered reversal bounded Turing machines are one-tape machines. Therefore we omit the subscript 1 (denoting the number of tapes) in $\mathrm{DREV}_1(\cdots)$, $\mathrm{NREV}_1(\cdots)$, and $\mathrm{AREV}_1(\cdots)$.

Different connections between reversal complexity and time and space complexities have been found. It is well known that $\mathrm{NSPACE}(S(n)) = \mathrm{NREV}(S(n))$ for all $S(n) \geqq n$ (Chytil [6]). The corresponding result for deterministic classes is $\mathrm{DSPACE}(\mathrm{Pol}\, S(n)) = \bigcup_k \mathrm{DREV}_k(\mathrm{Pol}\, S(n))$ for all $S(n) \geqq n$ (Hong [12]). For alternating machines we have $\mathrm{ASPACE}(S(n)) \subseteq \mathrm{AREV}(\log^* S(n))$ for all $S(n) \geqq n$ (Liśkiewicz, Loryś, and Piotrów [13]).

The next important question that was investigated for reversal complexity is a hierarchy problem. It was proved by Hartmanis [11] that for one-tape deterministic Turing machines with one-way input tape incrementing the number of reversals by one leads to a strictly larger class. On the other hand, Fischer [8] proved that for one-tape deterministic Turing machines $\mathrm{DREV}(R(n)) = \mathrm{DREV}(c \cdot R(n))$ for each

* Received by the editors August 11, 1986; accepted for publication (in revised form) December 19, 1988. This research was supported by the Polish government under program RP.I.09.

† Institute of Computer Science, University of Wrocław, ul. Przesmyckiego 20, 51-151 Wrocław, Poland.

constant c. Chan [3] obtained some hierarchy results for multicounter machines. These results were refined by Duriš and Galil [7].

Our main results concern the hierarchy of AREV classes. First, we investigate what is the lower bound for recognizing nonregular languages. Recall that

$$\log^*(n) = \min\{k: \underbrace{\log(\log\cdots(\log n)\cdots)}_{k \text{ times}} < 1\}.$$

We show that if $\lim(\log^* n - R(n)) = +\infty$, then $\mathrm{AREV}(R(n))$ is included in the class of regular languages (hence both are equal). This provides a striking characterization of regular languages, since it follows from [13] that all languages recognized by

$$\underbrace{2^{2^{\cdot^{\cdot^{2^n}}}}}_{c \text{ times}}\text{-space bounded ATMs}$$

belong to $\mathrm{AREV}(\log^* n)$, where c is a constant. In the case of deterministic machines, the corresponding lower bound for recognizing nonregular languages is equal to $\log n$ (Hartmanis [10]), and in the case of nondeterministic machines it is $\log\log n$ (compare Chytil [6]).

The function $\log^* n$ is a boundary above which a hierarchy theorem for AREV classes holds. Namely, if a function R is reversal-constructible and $\lim(R(n) - Z(n)) = +\infty$, then $\mathrm{AREV}(Z(n)) \subsetneq \mathrm{AREV}(R(n))$. On the other hand, we prove that for each machine the number of reversals can be decremented by a constant.

In the last section we deal with time-bounded ATMs. It is shown that $\mathrm{ATIME}_k(T(n)) \subseteq \mathrm{ATIME\text{-}REV}(T(n), \log^* T(n))$.

2. A lower bound for recognizing nonregular languages. For the sake of completeness let us recall the definition of the AREV classes.

DEFINITION 2.1. (a) We say that an alternating machine M *accepts* a word w making *at most r reversals* if there is an accepting computation tree for w such that along each computation path, M makes no more than r reversals.

(b) We say that $L \in \mathrm{AREV}(R(n))$ if there is a one-tape ATM, M, recognizing L, such that for each $w \in L$ the machine M accepts w making at most $R(|w|)$ reversals. Sometimes we say that $M \in \mathrm{AREV}(R(n))$ to indicate that the machine M has the properties stated above.

We commence with a simple observation.

DEFINITION 2.2. We say that M, a one-tape Turing machine, is *normal* if M makes reversals only at the ends of the previously used portion of the tape. That is between each two reversals the head of M moves through all cells that have been visited so far.

LEMMA 2.3. *For each machine* $M \in \mathrm{AREV}(R(n))$ *there is an equivalent machine* $M' \in \mathrm{AREV}(R(n))$ *that is normal. The same holds for* NREV *and* DREV *classes.*

Sketch of the proof. M' closely simulates M. During the simulation a portion of the tape, which has been visited by M, is indicated by two end markers. The idea is that if M makes a reversal between end markers then M' interrupts the simulation, leaves a special marker in the place where M makes the reversal, and without a reversal moves until the end marker is found. Then M' makes a reversal and moves to the place where the special marker was left. Then M' resumes the simulation of M. Clearly, such described M' makes the same number of reversals as M. □

In the next lemma we use *codes* of ATMs. Each such *code* is a set of quintuples characterizing a machine. If a machine uses s states, then the number of distinct

quintuples does not exceed s^2 multiplied by a constant depending on the alphabet used by the machine. Hence if we encode states as numbers written in binary, then the total length of the code is not greater than $c \cdot s^2 \cdot \log s$, where c is a constant depending on the alphabet.

Recall that for $a, b \in N$ by $a \dot{-} b$ we mean $\max(0, a-b)$. In the next lemma we give a method of reducing the number of reversals by a constant. Moreover, if a machine $M \in \mathrm{AREV}(R(n))$ is given then we not only find an equivalent machine $M' \in \mathrm{AREV}(R(n) \dot{-} 1)$, but we also indicate that a code of M' can be easily derived from a code of M. Let us describe how M' is constructed. By Lemma 2.3 we may assume that M is normal. Until the last two reversals M' operates exactly as M. After that, the head of M moves in one direction, makes a reversal, and then comes back. The head of M' cannot come back, hence sufficient information must be derived to avoid the last reversal. The idea is similar to that of replacing a two-way finite automaton by a one-way automaton. However, unlike in the case of finite automata, ATM can change the content of the tape, so we are not able to decrement at once the number of reversals by more than one.

LEMMA 2.4 (technical). *There is a one-tape deterministic Turing machine P such that for a given input word w, a code of a normal* ATM *M, the machine P computes a code of a machine M' such that*

(1) *M and M' recognize the same language;*

(2) *M' is normal;*

(3) *if $M \in \mathrm{AREV}(R(n))$, then $M' \in \mathrm{AREV}(R(n) \dot{-} 1)$; and*

(4) *if s and t denote the number of states of M and M', respectively, then $t = 2 + 8s + 2s \cdot 2^s$.*

Moreover, during the computation P uses no more space than needed to write a code of M'.

Proof. First we describe the algorithm executed by M'. The simulation of M is divided into the following stages:

Stage 1. M' operates exactly as M unless an attempt is made to make a reversal. Then M' splits existentially into two machines that start the execution of Stage 2 and Stage 3, respectively.

Stage 2. M' makes a reversal exactly as M does. Then M' returns to Stage 1.

Stage 3. Let w be a word written on the tape. Assume that the head of M is at the left end of w (if M is at the right end of w, the algorithm is essentially the same). M' makes the reversal, as M does, and transforms into a one-way alternating automaton F. The automaton F verifies whether M accepts w without more than one reversal. Two activities are performed simultaneously by F: the first one is a direct simulation of M while it moves to the right, the second one is getting information necessary to avoid moving to the left.

Let us say more about F. Let Q be the set of states of M. Each state of F has the form $[p_r, Z]$, where $p \in Q$ and $Z \subseteq Q$. Assume that $w = uav$ (a is a character) and M has moved from the left end of w to the cell occupied by a. Assume that during this, M has replaced u by u'. If M arrives at a in state p, then the corresponding state of F is $[p_r, Z]$ where for $q \in Q$ we have:

$$q \in Z \Leftrightarrow M \text{ can accept } u' \text{ making no moves to the right if started in state } q \text{ with the head at the right end of } u'.$$

Of course when M makes the reversal at the right end of w, then F has only to consult the current set Z to decide whether M will accept without further reversals. The major

problem is how to update the set Z when M (and hence also F) moves to the right. If M makes a stationary move, then Z remains unchanged. So assume that the head of M moves to the right, leaving some b in place of a. Then Z is replaced by Z', which is generated as follows.

Let $q \in Q$. We start an execution of M in state q on a one-letter word b. We put $q \in Z'$ if and only if there is a computation tree such that on each path:

(i) M makes only stationary moves and no more than one move to the left, and

(ii) M either leaves the cell occupied in the beginning by b in a state from Z, or stops in the meantime in an accepting state.

It is clear that M' fulfills the conditions (1)–(3). To verify the last one, we have to write down all states of M'. Since M' uses three stages of computation, there are distinct states and quintuples connected with each stage.

1. *States and quintuples corresponding to Stage* 1. For each $q \in Q$ we have two corresponding states, q_l, q_r. During the simulation, the state q_l will be used instead of q while M moves to the left, and q_r will be used while M moves to the right. Also, if q is universal (existential), then q_l, q_r are universal (existential), too. If q is an initial state of M, then q_r is an initial state of M'.

Let C and C' denote the sets of quintuples of M and M', respectively. Then for each $q \in Q$ and each symbol a, the following quintuples belong to C':

$(q_r, a, q'_r, a', 1)$ for each q', a' such that $(q, a, q', a', 1) \in C$ (1 denotes a move to the right);

$(q_r, a, q''_r, a'', 0)$ for each q'', a'' such that $(q, a, q'', a'', 0) \in C$;

$(q_r, a, R_{q_r}, a, 0)$ if there are p, b such that $(q, a, p, b, -1) \in C$;

$(q_l, a, q'_l, a', -1)$ for each q', a' such that $(q, a, q', a', -1) \in C$;

$(q_l, a, q''_l, a'', 0)$ for each q'', a'' such that $(q, a, q'', a'', 0) \in C$;

$(q_l, a, R_{q_l}, a, 0)$ if there are p, b such that $(q, a, p, b, 1) \in C$.

The states R_{q_r} and R_{q_l} are used in the situation when M makes a reversal. Both are existential and cause exiting from Stage 1.

2. *States and quintuples corresponding to Stage* 2. For each q and a the following quintuples belong to C':

$(R_{q_r}, a, P_{q_r}, a, 0)$ and $(R_{q_l}, a, P_{q_l}, a, 0)$;

$(P_{q_r}, a, q'_l, a', -1)$ for each q', a' such that $(q, a, q', a', -1) \in C$;

$(P_{q_l}, a, q''_r, a'', 1)$ for each q'', a'' such that $(q, a, q'', a'', 1) \in C$.

The states P_{q_r} and P_{q_l} are universal (existential) if q is universal (existential). At first look, it seems to be redundant to introduce the states R_{q_l}, R_{q_r}, P_{q_l}, P_{q_r}. However, R_{q_l}, R_{q_r} must be existential because they enable nondeterministic transition to Stage 2 or 3. After this transition, a correct mode of M, universal or existential, must be regained. The states P_{q_l}, P_{q_r} are used for this purpose. They cause an immediate return to Stage 1.

3. *States and quintuples corresponding to Stage* 3. The reversal of M' and transformation into a one-way alternating automaton F are caused by the following quintuples (for each q and a):

$(R_{q_r}, a, S_{q_r}, a, 0)$ and $(R_{q_l}, a, S_{q_l}, a, 0)$;

$(S_{q_r}, a, [q'_l, \mathrm{Beg}_r(a')], a', -1)$ for each q', a' such that $(q, a, q', a', -1) \in C$;

$(S_{q_l}, a, [q''_r, \mathrm{Beg}_l(a'')], a'', 1)$ for each q'', a'' such that $(q, a, q'', a'', 1) \in C$.

The states S_{q_r} and S_{q_l} are universal (existential) if q is universal (existential). Now $[q_l, \mathrm{Beg}_r(a)]$ and $[q_r, \mathrm{Beg}_l(a)]$ are the first examples of the states connected with the machine F. As we have mentioned, each state of F has two components, the first one is of the form q_l, q_r, the second one is a subset of Q. The mode (universal or existential) of such a composed state is determined by the mode of q. Now let us say how the sets $\mathrm{Beg}_l(a)$, $\mathrm{Beg}_r(a)$ are determined. For $q \in Q$ we have $q \in \mathrm{Beg}_r(a)$ if starting in state q, the machine M can build an accepting tree over the one-letter word a. Moreover, during this computation moves to the left are prohibited. The set $\mathrm{Beg}_l(a)$ is defined in the same way, but no move to the right is allowed. During one particular execution, F can move only to the right or only to the left. There are separate sets of quintuples corresponding to these two options. Since these sets differ inessentially, we describe only one of them by assuming that F moves to the right.

For a symbol b and $Z \subseteq Q$ we define a set $\mathrm{Next}_r(b, Z) \subseteq Q$ such that $q \in \mathrm{Next}_r(b, Z)$ if and only if there exists a subtree T of the full computation tree of the machine M over the one-letter word b such that

(i) assuming that all states from Z are accepting (besides the regular accepting states of M) T is an accepting tree,

(ii) q is the state in the root of T, and

(iii) on every path of T, either

(a) M makes only stationary moves and stops in a regular accepting state,

or

(b) after some number of stationary moves, M makes one move to the left and enters a state from Z or a regular accepting state.

Now we are ready to list the quintuples of F. For each state q, $Z \subseteq Q$, and symbol a the following quintuples are quintuples of F.

$([q_r, Z], a, [q'_r, \mathrm{Next}_r(a', Z)], a', 1)$ for each q', a' such that $(q, a, q', a', 1) \in C$;

$([q_r, Z], a, [q''_r, Z], a'', 0)$ for each q'', a'' such that $(q, a, q'', a'', 0) \in C$,

$([q_r, Z], a, \mathit{accept}, a, 0)$ if q is *accept*,

$([q_r, Z], a, \mathit{accept}, a, 0)$ if there are p' and a' such that $(q, a, p', a', -1) \in C$ and $p' \in Z$,

$([q_r, Z], a, \mathit{reject}, a, 0)$ if there are p'' and a'' such that $(q, a, p'', a'', -1) \in C$ and $p'' \notin Z$.

The states *accept* and *reject* are final and accepting or rejecting, respectively.

The reader may check that M' defined by the above quintuples works exactly as it was described at the beginning of the proof. Moreover, it can be easily checked that we have used $2+8s+2s \cdot 2^s$ states for M' if s denotes the number of states of M.

The method of generating the quintuples of M' is so simple that a code of M' can be easily derived from a code of M by some deterministic machine, which uses only the space needed for its input and the resulting code. □

COROLLARY 2.5. $\mathrm{AREV}(R(n)) = \mathrm{AREV}(R(n) \dot{-} c)$ *for all* R *and* $c \in N$. *Even more, if* M *is a normal machine and* $M \in \mathrm{ATIME\text{-}REV}(T(n), R(n))$, *then there is an equivalent machine* $M' \in \mathrm{ATIME\text{-}REV}(T(n), R(n) \dot{-} c)$.

Proof. In the proof of Lemma 2.4 for a given normal machine $M \in \mathrm{AREV}(R(n))$ we have constructed $M' \in \mathrm{AREV}(R(n) \dot{-} 1)$. It easily follows from the construction that if M makes $T(n)$ steps to accept, then M' makes no more than $T(n)+2R(n)$

steps to accept. But $R(n) \leqq T(n)$, so the result follows for $c = 1$. To get the rest, a straightforward induction can be applied. □

THEOREM 2.6. *For each language* $L \in \mathrm{AREV}(R(n))$ *there is a constant* c *such that*

$$L \in \mathrm{ASPACE}(\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{R(n)+c}).$$

Proof. Let M be an ATM recognizing L and making at most $R(n)$ reversals. We have to define ATM G which simulates M and works within the tape bounds stated in the theorem. Let w be a given input word. We describe how the machine G works.

Stage 1. G guesses existentially a number k such that $k = R(|w|)$. G writes a code of machine M on the work tape. Then using the algorithm from Lemma 2.4, G finds codes of machines $M_0 = M, M_1, M_2, \cdots, M_k$ equivalent to M of reversal complexity $R(n)$, $R(n) \dot{-} 1$, $R(n) \dot{-} 2, \cdots, R(n) \dot{-} k$, respectively.

Stage 2. G simulates M_k on the word w.

Now we check that the space used by G does not exceed the stated bound. By Lemma 2.4 the space used for generating a code of M_k is not larger than the code itself. In Stage 2 the machine M_k makes $R(|w|) \dot{-} k = 0$ reversals, so in fact M_k works as an alternating finite automaton, and hence no additional space is required other than a space needed to record a current state. So we have to find the number of states of M_k. Suppose x is the number of states of M. Let $\eta_x(i)$ be the number of the states of M_i. By Lemma 2.4 we have

$$\eta_x(i+1) = 2 + 8\eta_x(i) + 2\eta_x(i) \cdot 2^{\eta_x(i)}, \quad \text{also } \eta_x(0) = x.$$

CLAIM. *There is a constant* c *(depending on* x*) such that*

$$2\eta_x(i) \leqq \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{i+c \text{ times}}$$

for all i*. Hence also*

$$\eta_x(i) \leqq \underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{i+c \text{ times}}.$$

Proof of the Claim. Without loss of generality we may assume that $x \geqq 5$. We prove the Claim by induction on i. Let c be such that

$$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{c \text{ times}} \geqq 2 \cdot x = 2 \cdot \eta_x(0).$$

Assume now that

$$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{c+i \text{ times}} \geqq 2 \cdot \eta_x(i).$$

Then

$$\underbrace{2^{2^{\cdot^{\cdot^{2}}}}}_{c+i+1 \text{ times}} \geqq 2^{2 \cdot \eta_x(i)} = 2^{\eta_x(i)} \cdot 2^{\eta_x(i)}.$$

On the other hand, since $\eta_x(i) \geqq x \geqq 5$ we have

$$2\eta_x(i+1) = 2 \cdot (2+8\eta_x(i)+2\eta_x(i) \cdot 2^{\eta_x(i)}) = 2^{\eta_x(i)} \cdot 2 \cdot \left(2\eta_x(i)+\frac{8\eta_x(i)+2}{2^{\eta_x(i)}}\right)$$
$$\leqq 2^{\eta_x(i)} \cdot 2^{\eta_x(i)}.$$

So

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{c+i+1 \text{ times}} \geqq 2\eta_x(i+1),$$

as required. □

Now by the Claim and the considerations that precede the Claim, it follows that G uses the space of size

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{k+c \text{ times}}$$

for inputs of the length n. So

$$G \in \text{ASPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)+c}\right). \qquad \square$$

Remark. It is proved by Liśkiewicz, Loryś, and Piotrów [13] that $\text{ASPACE}(S(n)) \subseteq \text{AREV}(\log^* S(n))$ for $S(n) \geqq n$. In a view of Theorem 2.6, the function $\log^* S(n)$ cannot be replaced by a smaller one. Indeed, let $S(n) \geqq 2^{2^n}$. If $\lim(\log^* S(n) - Z(n)) = +\infty$ and $\text{ASPACE}(S(n)) \subseteq \text{AREV}(Z(n))$, then we would have

$$\text{ASPACE}(S(n)) \subseteq \text{AREV}(Z(n)) \subseteq \text{ASPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{Z(n)+T(n)}\right),$$

where $T(n)$ is any function such that $\lim T(n) = +\infty$. Then also

$$\text{ASPACE}(S(n)) \subseteq \text{ASPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{\log^* S(n)-3}\right) \subseteq \text{ASPACE}(\log\log\log S(n))$$
$$\subseteq \bigcup_c \text{DTIME}(c^{\log\log\log S(n)}) \subseteq \text{DTIME}(\log S(n)).$$

If S is time-constructible, then $\text{DTIME}(\log S(n)) \subsetneq \text{DTIME}(S(n)) \subseteq \text{ASPACE}(S(n))$. Contradiction: For $n \leqq S(n) \leqq 2^{2^n}$ the Remark follows by Theorem 2.7. □

As another corollary of Theorem 2.6, we obtain the lower bound for recognizing nonregular languages.

THEOREM 2.7. *If* $\lim(\log^* n - Z(n)) = +\infty$, *then* $\text{AREV}(Z(n)) \subseteq \text{Reg}$, *where* Reg *denotes the class of regular languages.*

Before the proof of Theorem 2.7, notice that from the inclusion $\text{ASPACE}(S(n)) \subseteq \text{AREV}(\log^* S(n))$ and Corollary 2.5 it follows that, for every constant c, the class

$$\text{ASPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2^n}}}}}}_{c \text{ times}}\right)$$

is included in $\text{AREV}(\log^* n)$. Hence $\text{AREV}(\log^* n)$ is a very large class. However, even minor decreasing of the function $\log^* n$ leads to the class of regular languages.

Proof of Theorem 2.7. Suppose $L \in \mathrm{AREV}(Z(n))$. By Theorem 2.6 there is a constant c such that

$$L \in \mathrm{ASPACE}(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{Z(n)+c}).$$

Since $\log^* n - 4 > Z(n) + c$ for almost all n, so

$$L \in \mathrm{ASPACE}(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{\log^* n - 4}) \subseteq \mathrm{ASPACE}(\log\log\log\log n).$$

It was proved by Alberts [1] that for each function $S(n) = o(\log\log\log n)$ the class $\mathrm{ASPACE}(S(n))$ contains only regular languages. Hence the theorem follows. □

For the sake of completeness we now examine what is the lower reversal bound for recognizing nonregular languages if we use deterministic and nondeterministic machines. First we show that in the case of deterministic Turing machines this bound is $\log n$.

PROPOSITION 2.8. *For each constant c the language $L = \{a^n b^n : n \in N\}$ belongs to* $\mathrm{DREV}(\log(n)/c)$. *Hence* $\mathrm{DREV}(\log(n)/c) \not\subseteq \mathrm{Reg}$.

Proof. The following deterministic algorithm using $\log n$ reversals recognizes L.

Stage 1. The head moves to the right end of the input checking if it is of the form $a^n b^m$.

Stage 2. The head changes its direction. While it moves through the word, it erases half of the a's and b's still on the tape. Namely, the first a, the third a, the fifth $a\ldots$ are erased and the first b, the third b, the fifth $b\ldots$ are erased. When the head reaches the opposite end of the word, the following can be said.

(i) If there are no symbols left, then $n = m$ and the word should be accepted, and

(ii) If the last met a was not erased and the last b was erased, then $m \neq n$ and the word should be rejected. This also holds if the last a was erased but b was not.

These cases are checked directly. If none of them holds, then the machine returns to the beginning of Stage 2.

It is easy to see that $\log n$ reversals would suffice. So the language $\{a^n b^n : n \in N\}$ is in $\mathrm{DREV}(\log n)$. Since $\mathrm{DREV}(R(n)) = \mathrm{DREV}(c \cdot R(n))$ for all R and $c \in N$ (Fischer [8]), the proposition follows. □

PROPOSITION 2.9 (Hartmanis). $\mathrm{DREV}(o(\log n)) \subseteq \mathrm{Reg}$.

Proof. Let $M \in \mathrm{DREV}(R(n))$ for some function $R(n) = o(\log n)$. We show that the language L recognized by M is regular. It follows from Lemma 2.3 that we may assume that M is normal. Let t be the number of states of M and $c > \log t$. There exists a constant n_0 such that M makes no more than $\log(n)/c$ reversals for each word from L of length $n \geqq n_0$.

Let $m > \max_{i \leqq n_0} R(i)$. We now show that each word from L can be accepted by at most m reversals. Assume the converse. Let $x \in L$ be the shortest word that requires m' reversals for some $m' > m$. Of course, $|x| > n_0$. Let us consider all crossing sequences between symbols of x. The total number of different crossing sequences does not exceed $t^{\log(|x|)/c} = |x|^{\log(t)/c} < |x|$. Therefore there are two different places in x that have the same crossing sequences. Let $x = uvw$ (u, v, w-nonempty words) and the crossing sequence between u and v is the same as that between v and w. It is clear that uw and x are accepted by the same number of reversals. This is a contradiction, since $|uw| < |x|$.

We have shown that $L \in \mathrm{DREV}(m)$, but $\mathrm{DREV}(m) \subseteq \mathrm{Reg}$ by Theorem 2.7. □

Now we show that in the case of nondeterministic Turing machines the lower bound for recognizing nonregular languages is log log n. The results are given in the following two propositions.

PROPOSITION 2.10. *Suppose* $S(n) = o(\log\log n)$. *Then* $\mathrm{NREV}(S(n)) \subseteq \mathrm{Reg}$.

Proof. Recall that for each function $S(n)$, $\mathrm{NREV}(S(n)) \subseteq \mathrm{NSPACE}(S(n))$ [6]. So if $S(n) = o(\log\log n)$, then the result holds since $\mathrm{NSPACE}(S(n)) = \mathrm{Reg}$ (see Alberts [1]). □

LEMMA 2.11. *There exists constant* d *such that for all* $n, m \in N$ *if* $n \neq m$, *then* $n \not\equiv m \bmod (p)$ *for some prime* $p \leqq d \cdot \log(\max(n, m))$.

Proof. For $x \in N$ we set

$$\Theta(x) = \sum_{\substack{p \leqq x \\ p \text{ prime}}} \ln p.$$

There exists a constant c such that $\Theta(x) \geqq c \cdot x$ (see Prachar [15]). From that we can simply get

$$\prod_{\substack{p \leqq d \cdot \log x \\ p \text{ prime}}} p \geqq x$$

for some constant d. Now let $n > m$. If $m \equiv n \pmod p$ for all $p \leqq d \cdot \log n$, then by the Chinese Reminder Theorem

$$m \equiv n \left(\bmod \prod_{\substack{p \leqq d \cdot \log n \\ p \text{ prime}}} p \right).$$

But

$$m, n < \prod_{\substack{p \leqq d \cdot \log n \\ p \text{ prime}}} p,$$

so $m = n$, a contradiction. □

PROPOSITION 2.12. *Let* $L = \{a^n b^m : n \neq m\}$. *Then* $L \in \mathrm{NREV}(\log\log(n)/c)$ *for each* $c \in N$. *Hence* $\mathrm{NREV}(\log\log(n)/c) \not\subseteq \mathrm{Reg}$.

Proof. The following algorithm recognizes L.

Stage 1. The machine checks if the input word is of the form $a^n b^m$. Simultaneously, the machine nondeterministically puts some number of symbols $\not\equiv$ on the second track. They are used to divide $a^n b^m$ into smaller blocks, say, $a^n b^m = w_1 w_2 \cdots w_l u_1 u_2 \cdots u_t$, where each w_i is a subword of a^n and each u_j is a subword of b^m.

Stage 2. The machine accepts if the following conditions are checked true:

(i) $|w_1| = |w_2| = \cdots = |w_{l-1}| = |u_1| = |u_2| = \cdots = |u_{t-1}|$,

(ii) $|w_l| \neq |u_t|$ and $|w_l|, |u_t| < |w_1|$.

Essentially the same algorithm as in Proposition 2.8 is used. The head moves several times between the ends of the input word. During each round the head erases half of the symbols from each block, namely, the first symbols, the third ones, and the fifth ones $\cdots$. The same procedure as in Proposition 2.8 is used to accept or reject. We leave all remaining details to the reader.

Notice that the machine makes at most $\log(w) + 1$ reversals, where w is the length of the longest block.

Now let $a^n b^m \in L$ and p be the number given by the Claim. We may assume that all blocks, except w_l and u_t, have the length p, and $|w_l|, |u_t| < p$. Then $n \equiv |w_l| \pmod p$

and $m \equiv |u_t| \pmod{p}$, so $|w_l| \neq |u_t|$. Hence the machine accepts $a^n b^m$ and uses no more than $\log p \leqq \log(d) + \log\log(\max(n, m))$ reversals to accept.

It is easy to check that $\mathrm{NREV}(S(n)) = \mathrm{NREV}(c \cdot S(n))$ for all S and $c \in N$. So the proposition holds. □

3. A hierarchy for reversal-bounded ATMs.

DEFINITION 3.1. A function f is alternating reversal-constructible if there is a one-tape alternating machine M that recognizes the language $L_f = \{0^n 1^{f(n)} : n \in N\}$ using at most $f(n)$ reversals.

Note that the class of alternating reversal-constructible functions is very wide. For instance, it contains log*, log, all polynomials, exponential functions $\cdots$. Moreover, for many f the language L_f can be recognized with no more than $\log^*(n)$ reversals.

PROPOSITION 3.2. *Suppose f is alternating reversal-constructible. Then for some constant c the function*

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{f(n)+c}$$

is deterministic space-constructible.

Proof. Let f be alternating reversal-constructible. Slightly modifying the proof of Theorem 2.6, we obtain that the language L_f is recognized by some ATM M', that accepts words $0^n 1^{f(n)}$ within space

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{f(n)+c'}$$

where c' is a constant. Thus L_f is recognized also by some deterministic machine M'' that needs at most

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{f(n)+c'+2}$$

cells to accept words $0^n 1^{f(n)}$.

We can construct a deterministic machine M that for a given input 0^n finds $f(n)$ and then marks

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{f(n)+c'+2}$$

cells on the work tape. The number $f(n)$ is found by the following algorithm.

$i := -1$; *found* := *false*;
repeat
  $k := 0$; $i := i+1$;
  while $k \leqq i$ and *not found* do
    if M'' accepts $0^n 1^k$ using i cells then *found* := *true* else $k := k+1$;
until *found*

Let $c = c'+2$. It is easy to see that the number of cells needed by M to find $f(n)$ is bounded by

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{f(n)+c}.$$

Then the machine M in a simple manner marks

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{f(n)+c} \text{ cells.}$$

□

THEOREM 3.3. *Suppose R and Z are functions such that*

(i) $\lim (R(n)-Z(n))=+\infty$,

(ii) $R(n)\geqq \log^* n$,

(iii) *R is alternating reversal-constructible.*

Then $\mathrm{AREV}(Z(n))\subsetneq \mathrm{AREV}(R(n))$.

Proof. By Theorem 2.6 we have

$$\mathrm{AREV}(Z(n))\subseteq \mathrm{ASPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)}\right).$$

It is easy to see that $R(n)+3$ is alternating reversal-constructible, so by Proposition 3.2 for some constant $c\geqq 3$ the function

$$\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)+c}$$

is deterministic space-constructible. Hence we have

$$\mathrm{ASPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)}\right)\subseteq \mathrm{DTIME}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)+2}\right)\subseteq \mathrm{DSPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)+2}\right)\subsetneq \mathrm{DSPACE}\left(\underbrace{2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}}_{R(n)+c}\right)$$

$$\subseteq \mathrm{AREV}(R(n)+c).$$

So $\mathrm{AREV}(Z(n))\subsetneq \mathrm{AREV}(R(n)+c)$. By Corollary 2.5 $\mathrm{AREV}(R(n)+c)=\mathrm{AREV}(R(n))$; hence, the result follows. □

4. Reversal complexity of time-bounded ATMs. In this section we shall prove that for ATMs the number of reversals can be reduced to some small number without loss of time.

If w is a string of the numbers -1, 0, and 1, say $w=a_1a_2\cdots a_n\in\{-1, 0, 1\}^*$, then $\sum w$ stands for $a_1+a_2+\cdots+a_n$. We prove the following technical proposition.

LEMMA 4.1. *There is an alternating Turing machine M working in linear time and making* $\log^* n$ *reversals such that for given words w, $u\in\{-1, 0, 1\}^*$ the machine M checks if $u=\sum w$.*

Proof. We may assume that inputs are given on the first track of the tape. The following recursive algorithm solves the problem.

For a given input word w if $|w|$ is small enough, then $\sum w$ is determined directly with no reversals. Otherwise, M executes the algorithm described below. Of course M chooses existentially between these two options. We shall prove that the total number of reversals made by M does not exceed $\log^* |w|-1$ and the time used is linear.

In the existential mode, M goes to the opposite side of w using tracks number 2, 3, 4, 5 to write certain sequences (see Fig. 1).

On the first track, w is left unchanged. On the second track, M puts the symbol $\#$ several times to divide the word w into smaller blocks (the last block may contain blanks from outside of w). Simultaneously, inside the ith block, on tracks number 3, 4, and 5, M writes binary numbers x_i, y_i, and z_i, respectively. Let w_i denote the part of w lying inside the ith block. The numbers written by M are to have the following

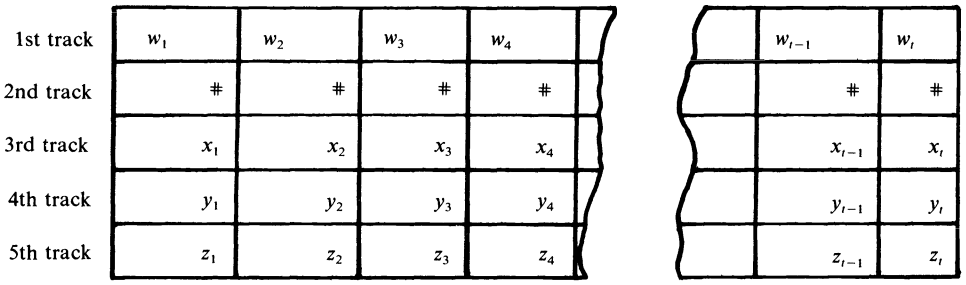


FIG. 1

properties.

(4.1) $\quad x_i = \sum w_i, \quad y_i = z_{i-1}, \quad z_i = y_i + x_i$ for all $i \leqq t$ and $y_1 = 0$.

The point is that from these equalities it follows that

$$z_i = \sum w_1 + \sum w_2 + \cdots + \sum w_i.$$

Clearly such numbers x_i, y_i, z_i exist, and the only trouble that could arise is when the blocks are too small. Then simply there is no room inside the blocks to write down the numbers z_i. We avoid this if each block has the length at least $\log|w|$. We may assume that each block has the length $\log|w|$.

Now M has to check that the equalities (4.1) hold. M makes a reversal and splits universally in two machines. The first one of them checks if $z_t = u$, the second one moves through the word and for each i creates universally the following machines:

(i) one checking if $y_i = z_{i-1}$,

(ii) one checking if $z_i = y_i + x_i$, and

(iii) one checking if $x_i = \sum w_i$.

The first machine makes $\log^*|y_i| < \log^*|w|$ reversals and works in time $|y_i|$ (compare Lemma 3.1 from [13]). The second machine makes at most one reversal (it depends on the direction from which the head has arrived) and works in time $|z_i|$. The third machine executes the same algorithm but now for a word of the length $\log|w|$. By the induction hypothesis this machine works in time $O(\log|w|)$ and makes no more than $\log^*(\log|w|) - 1$ reversals. Now we see that the total time used by M is linear. Also the number of reversals does not exceed $(\log^*(\log|w|) - 1) + 1 = \log^*|w| - 1$. □

Let us recall two useful algorithms from Liśkiewicz, Loryś, and Piotrów [13, Lem. 3.1].

LEMMA 4.2. *Let a word x, a binary number m, and a symbol b be given. There are* ATMs M_1 *and* M_2 *making at most* $\log^*|x|$ *reversals and working in time* $O(|x|)$ *such that*

(i) M_1 *determines whether* $m > |x|$, *and*

(ii) M_2 *checks whether the* mth *symbol of* x *is* b.

Now we are ready to prove the result we want. Recall that $\text{ATIME}_k(T(n))$ is the class of languages recognizable by k-tape ATMs in time $T(n)$. Similarly, $\text{ATIME-REV}(T(n), R(n))$ denotes the class of languages that can be recognized by one-tape ATMs that work in time $T(n)$ and make at most $R(n)$ reversals.

THEOREM 4.3. $\text{ATIME}_k(T(n)) \subseteq \text{ATIME-REV}(T(n), \log^* T(n))$.

Proof. Paul, Prauss, and Reischuk [14] showed that $\text{ATIME}_k(T(n)) \subseteq \text{ATIME}_1(T(n))$, so we may consider exclusively one-tape machines. Let M be such an ATM working in time $T(n)$. Admittedly, M makes a large number of reversals, so

we have to simulate M in some nonstraightforward way. The machine M', defined below, makes the simulation. The idea is to write a whole computation path. However, the full configurations of M cannot be used for this purpose. Even recording whole tape contents would require too much time. Roughly speaking, for each execution step of M, the machine M' puts on its tape the record $[a, b, m]$, where

- a is the symbol that the head of M reads,
- b is a new symbol left by M in this place, and
- $m \in \{-1, 0, 1\}$ and denotes the move of M (-1 for the move to the left, 0 for the stationary move, and 1 for the move to the right).

So after t steps of the simulation the tape contains a word of the form $x[a_1, b_1, m_1][a_2, b_2, m_2] \cdots [a_t, b_t, m_t]$, where x denotes the input word. Using its states, M' remembers the state of M. Now we describe the simulation in detail:

Stage 0. First of all, M' existentially marks the end of the tape that will be used. Then it returns to the right end of x. If q_0, the initial state of M, is existential, then M' goes to Stage 1.1. Otherwise, M' goes to Stage 1.2. From now on q_M denotes the current state of M remembered by M'. Let δ_M be the transition function of M.

Stage 1.1. M' chooses existentially a symbol a and $(p, b, m) \in \delta_M(q_M, a)$. Then M' changes q_M to p and puts the record $[a, b, m]$ on the tape. If p is existential, then M returns to the beginning of Stage 1.1. If p is universal, then M' splits universally and goes to Stages 1.2 and 2. If p is an accepting state, then M goes to Stage 2. If p is a rejecting state, then M' rejects.

Stage 1.2. M' chooses universally a symbol a and $(p, b, m) \in \delta_M(q_M, a)$ (we may assume that for each universal q, $\delta_M(q, a)$ is nonempty). Then M' changes q_M to p and puts the record $[a, b, m]$ on the tape. If p is existential, then M' creates existentially two machines going to Stages 1.1 and 3 (respectively). If p is an accepting state, then M' accepts. If p is a rejecting state, then M' goes to Stage 3. Now we have to consider the only remaining possibility when p is universal. It may happen that M' has reached the end of the tape marked during Stage 0. It means that the guessed computation path is not a path from an accepting computation tree. If at least one guess was not correct, we have to accept. If all guesses were correct, then it means that we are describing a valid computation path of M, and this path is longer than the allowed time of computation. Then we have to reject. So in either case, machine M' goes to Stage 3. In the cases when the endmarker was not reached and p was universal, M' returns to the beginning of Stage 1.2.

Let us put a few lines of informal comments before the next stages. In Stages 1.1 and 1.2 the records of the form $[a, b, m]$ are guessed. Of course, nothing prevents M' from incorrectly guessing a. In Stages 2 and 3 we check the guesses. The checking is initiated at the moment when M changes its mode. Consider such a situation. Then the tape of M' contains some $x[a_1, b_1, m_1][a_2, b_2, m_2] \cdots [a_t, b_t, m_t]$. Let $q_1, \cdots, q_t$ be the states of M corresponding to the above records. Let us assume that q_t is existential and q_s is the last universal state. In fact, it would suffice to check that $a_{s+1}, \cdots, a_t$ are correct, since $a_1, \cdots, a_s$ have been checked previously. The algorithm presented here checks all of them but this is nonessential. Since M' has to check that the guesses are correct and that the further simulation leads to acceptance, M' splits universally into two machines, one making further simulation and one checking the guesses. Now let us assume that q_t is universal and so are $q_{s+1}, \cdots, q_{t-1}$. Since $a_{s+1}, \cdots, a_t$ were written in universal mode, we must accept when either one of the guesses was not correct or the further simulation leads to acceptance. Hence in this case M' must split existentially.

Stage 2. M' enters this stage with some $x[a_1, b_1, m_1][a_2, b_2, m_2] \cdots [a_t, b_t, m_t]$ on the tape. The machine now checks if the symbols $a_1, a_2, \cdots, a_t$ were guessed correctly. For this purpose M' splits universally and checks the correctness of each a_j. Let C be the cell at which the head of M stands during the round corresponding to $[a_j, b_j, m_j]$. To determine whether a_j was guessed correctly, M' either has to find the symbol left in C during the previous visit of M in C or to determine that C has never been visited so far. In the last case an appropriate symbol from the input tape should be found. The decision of which case occurs is made existentially and M' enters Substage 2.1 or 2.2. Notice that to determine a position of the head of M, say, at the round corresponding to $[a_i, b_i, m_i]$, it suffices to compute $\sum_{l=1}^{i-1} m_l$. At this round the head of M stands at C if and only if $\sum_{l=i}^{j-1} m_l = 0$.

Substage 2.1 (the case when C has not been visited so far). M' generates universally $j-1$ machines, each of them checking for some $i<j$ that $\sum_{l=i}^{j-1} m_l \neq 0$. Then M guesses $m = \sum_{l=1}^{j-1} m_l$ and splits universally. The first machine checks if m was guessed correctly and uses the algorithm from Lemma 4.1 for this purpose. The second machine again splits, but now existentially, and checks one of the following:

$$m < 1 \quad \text{and } a_j \text{ is a blank,}$$

$$\text{or } m > |x| \quad \text{and } a_j \text{ is a blank,}$$

$$\text{or } 1 \leqq m \leqq |x| \quad \text{and the } m\text{th symbol of } x \text{ is } a_j.$$

Notice that in each of these cases we can use an algorithm working in time $O(|x|)$ and making no more than $\log^* |x|$ reversals. The first case is straightforward. For the two remaining ones we can use the algorithms from Lemma 4.2.

Substage 2.2. Using the algorithm from Lemma 4.1, the machine M' checks that there is $i \leqq j-1$ such that

$$b_i = a_j \wedge \sum_{l=i}^{j-1} m_l = 0 \wedge \forall s \quad \left(i < s < j \Rightarrow \sum_{l=s}^{j-1} m_l \neq 0 \right).$$

Stage 3. This stage is the same as Stage 2 except that we have to accept when the guesses were not correct and to reject otherwise.

It is quite easy to verify that M' recognizes the same language as M. To show that M' satisfies needed complexity requirements, consider one computation path in an accepting tree. Before M' leaves Stage 1 no reversal is made. The time used also does not exceed $T(n)$. After leaving Stage 1 the machine executes either Stage 2 or Stage 3. Moreover, in either case it is made only once. Now consider what happens in Stage 2 or Stage 3. Most costly is the use of the algorithms from Lemma 4.1 and Lemma 4.2, but the words to which we apply these algorithms have lengths no greater than $T(n)$, so the time used is no greater than $O(T(n))$ and no more than $\log^* T(n)$ reversals are made.

By Corollary 2.5 the total number of reversals made in Stage 2 can be reduced to $\log^* T(n) - 1$ with no increase of time. Hence the total number of reversals does not exceed $\log^* T(n)$ and the theorem follows. □

By careful examination of the proof of Theorem 4.3, one can see that during the simulation the number of alternations does not increase significantly. If $A(n)$ is the number of alternations made by M, then M' makes no more than $A(n) + \log^* T(n)$ alternations. To get the same result for multitape ATMs, we must generalize slightly the presented proof (or re-prove the theorem of Paul, Prauss, and Reischuk). Simply each record would contain a triple of the form $[a, b, m]$ separately for each tape of

M. Finally, we have the result, which can be abbreviated as

$$\text{ATIME-ALT}_k\,(T(n), A(n)) \subseteq \text{ATIME-REV-ALT}\,(T(n), \log^* T(n), A(n)+\log^* T(n)).$$

Acknowledgment. We wish to thank Professor L. Pacholski for his support and encouragement.

REFERENCES

[1] M. ALBERTS, *Space complexity of alternating Turing machines*, in Fundamentals of Computation Theory, L. Budach, ed., Lecture Notes in Computer Science, 199, Springer-Verlag, Berlin, New York, 1985, pp. 1-7.

[2] B. BAKER AND R. V. BOOK, *Reversal-bounded multipushdown machines*, J. Comput. System Sci., 8 (1974), pp. 315-322.

[3] T.-H. CHAN, *Reversal complexity of counter machines*, Proc. 13th Annual ACM Symposium on the Theory of Computing, 1981, pp. 147-157.

[4] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[5] M. CHROBAK, *A note on reversal-bounded multipushdown machines*, Inform. Process. Lett., 19 (1984), pp. 179-180.

[6] M. P. CHYTIL, *On complexity of nondeterministic Turing machine computations*, in Mathematical Foundations of Computer Science, J. Becvár, ed., Lecture Notes in Computer Science, 32, Springer-Verlag, Berlin, New York, 1975, pp. 199-205.

[7] P. DURIŠ AND Z. GALIL, *On reversal bounded counter machines and on pushdown automata with a bound on the size of pushdown store*, Inform. and Control, 54 (1982), pp. 217-227.

[8] P. C. FISCHER, *The reduction of tape reversals for off-line one-tape Turing machines*, J. Comput. System Sci., 2 (1968), pp. 136-146.

[9] E. M. GURARI AND O. H. IBARRA, *Simple counter machines and number-theoretic problems*, J. Comput. System Sci., 19 (1979), pp. 145-162.

[10] J. HARTMANIS, *Computational complexity of one-tape Turing machine computations*, J. Assoc. Comput. Mach., 15 (1968), pp. 325-339.

[11] ———, *Tape-reversal bounded Turing machine computations*, J. Comput. System Sci., 2 (1968), pp. 117-135.

[12] J.-W. HONG, *On similarity and duality of computation*, Proc. 21st Annual IEEE Symposium on the Foundations of Computer Science, 1980, pp. 348-359.

[13] M. LIŚKIEWICZ, K. LORYŚ, AND M. PIOTRÓW, *On reversal bounded alternating Turing machines*, Theoret. Comput. Sci., 54 (1987), pp. 331-339.

[14] W. J. PAUL, E. J. PRAUSS, AND R. REISCHUK, *On alternation*, Acta Inform., 14 (1980), pp. 243-255.

[15] K. PRACHAR, *Primzahlverteilung*, Springer-Verlag, Berlin, Götingen, Heidelberg, 1957.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 222-245, April 1990

002

THE COMPLEXITY OF FILE TRANSFER SCHEDULING WITH FORWARDING*

JENNIFER WHITEHEAD†

Abstract. The file transfer scheduling problem was introduced and studied by Coffman, Garey, Johnson, and LaPaugh. This paper extends their model to include forwarding when no direct link exists between nodes. Several special cases of the problem, which were previously solvable by polynomial time algorithms, are shown to be NP-complete when forwarding is included. Other special cases are shown to continue to have polynomial time solutions in the forwarding model. All results assume the existence of a central controller.

Key words. network, scheduling, complexity

AMS(MOS) subject classifications. 68M10, 68R10, 90B35

1. Introduction. The problem of scheduling file transfers between nodes in a network to minimize overall finishing time was first introduced by Coffman et al. [4]. The model introduced in [4] represents an instance of the problem as a weighted undirected multigraph $G=(V, E)$, called a *file transfer graph.* The vertices of G correspond to the nodes of the network that are communication centers capable of communicating directly with every other center. Each vertex v in V is labeled with a positive integer $\alpha(v)$ that represents the number of communication modules at the node corresponding to v. It is assumed that each communication module may be used as a transmitter and as a receiver. Each edge $e \in E$ is labeled with a positive integer $w(e)$ that represents the amount of time needed for the transfer of a file corresponding to e (the file is transmitted between the nodes corresponding to the end vertices of e). The authors assume, in addition, that once the transfer of a file begins it continues without interruption. They show the general problem to be NP-complete but obtain polynomial time algorithms for various restrictions on the graph G. In the case where G is an odd cycle, Hakimi and Choi [2] obtain a polynomial time algorithm for the file transfer problem.

The question is raised in [4] as to whether their model extends to include the possibility of forwarding. If a communication center u wishes to send a file to v but no direct link exists, the file must be sent to one or more intermediaries who will then send it on to v. In this paper we study the file transfer problem when this type of forwarding is included.

The model we use is as described above, except that only files which may be sent directly between the centers that are its endpoints are represented with an undirected edge. Files that are sent to an intermediary will be represented as a directed edge indicating the direction of the transmission. The file may take several steps to reach its destination. We shall assume throughout the paper that schedules are constructed by a single central controller that, given the file transfer graph, constructs an overall schedule in advance. The forwarding will be assumed to be nonadaptive fixed routing where the destination rigidly determines the route. The route will be represented as a path of directed edges in the file transfer graph.

In § 2 of this paper we consider the case where all transmissions between nodes are assumed to be of a fixed length (i.e., $w(e)=1$). In contrast to the results in [4] we

* Received by the editors July 27, 1987; accepted for publication June 22, 1989.
† Department of Computer Science, Queens College, Flushing, New York, 11367.

find that when forwarding is included the problem is now NP-complete even for bipartite graphs where the degree $r(v)$ of each node is bounded. For trees the problem is NP-complete for $r(v)$ unbounded, but we obtain a polynomial time algorithm when $r(v)$ is bounded, assuming only single edges between nodes. The case of paths and cycles is examined separately, and a better time bound of $O(|E|)$ is obtained for the optimal algorithm.

In § 3 we assume the length of file transmissions may vary. In this instance, for paths and even cycles, the problem becomes NP-complete if multiple edges are allowed. For single edges we obtain an $O(|E| \cdot \log |E|)$ optimal algorithm in the case G is a path or cycle.

The file transfer problem has been studied by other authors who have assumed that the schedule may be interrupted (see [9], [10], [1]). If interruptions are allowed and transmitting and receiving modules are distinct, the problem has been studied in [3] and [8].

2. Complexity results for equal edge lengths.

2.1. Bipartite graphs. We shall show that the file transfer problem with forwarding (FTPF) is NP-complete even for bipartite graphs with fixed maximum degree = 3 at each node. We shall reduce the restricted timetable problem to FTPF. For convenience we state the timetable problem (TT) here (see [5]):

Given are: a set H of hours of the week, a set T of teachers, a set C of classes, a subset $A(c)$ of available hours for each class $c \in C$, a subset $A(t) \subseteq H$ of available hours for each teacher $t \in T$, and for each pair $(t, c) \in T \times C$ a number $R(t, c) \in Z_0^+$ of required teaching hours. The problem is to determine if there is a timetable for completing all tasks, i.e., a function $f: T \times C \times H \to \{0, 1\}$, where $f(t, c, h) = 1$ means teacher t teaches class c during period h such that:

(1) $f(t, c, h) = 1$ only if $h \in A(t) \cap A(c)$;

(2) For each $h \in H$ and $t \in T$ there is at most one $c \in C$ for each $f(t, c, h) = 1$;

(3) For each $h \in H$ and $c \in C$ there is at most one $t \in T$ for which $f(t, c, h) = 1$;

(4) For each pair $(t, c) \in T \times C$ there are exactly $R(t, c)$ values of h for which $f(t, c, h) = 1$.

A teacher t is called a k-teacher if $|A(t)| = k$. The teacher is tight if $|A(t)| = \sum_{c \in C} R(t, c)$, i.e., they must teach whenever they are available.

The restricted timetable problem (RTT) is that problem subject to the following restrictions:

(1) $|H| = 3$;

(2) $A(c) = H$ for all $c \in C$ (each class is always available);

(3) Each teacher is either a tight 2-teacher or a tight 3-teacher;

(4) $R(t, c) = 0$ or 1 for each $t \in T$, $c \in C$.

THEOREM 1. *File Transfer Scheduling with forwarding is* NP*-complete for G bipartite, maximum degree* = 3 *at each node, all edge lengths equal, no multiple edges and* $\alpha(v) = 1$ *for all* $v \in V$.

Proof. Given an instance I of RTT we construct a file transfer bipartite graph as follows. The nodes of G consist of a set T of teacher nodes one for each teacher, a set C of class nodes one for each class. In addition, we have sets D and S of nodes constructed by examining the restrictions on each teacher. Edges will go between $D \cup C$ and $S \cup T$ nodes only. Thus G is bipartite.

If teacher t is required to teach class c, draw an edge from t to c (representing a file transfer between t and c with no forwarding). Since each teacher is a 2- or 3-teacher, the maximum degree = 3 at each node. We add D and S nodes as follows for tight

2-teachers. Assume $H=\{t_1, t_2, t_3\}$. Suppose teacher t is a tight 2-teacher available during $\{t_1, t_2\}$. Then add nodes $s \in S$ and $d_1, d_2 \in D$ as shown in Fig. 1(a). Thus for deadline $d=3$, t is available only during the first two time intervals.

If t is a tight 2-teacher available during $\{t_1, t_3\}$ add nodes, $s_1, s_2, s_3, s_4, d_1, d_2, d_3$ as shown in Fig. 1(b). For deadline $d=3$, d_2 will be available to transmit to t only during the second time interval. Thus t is available for other transmissions only during the first and last intervals.

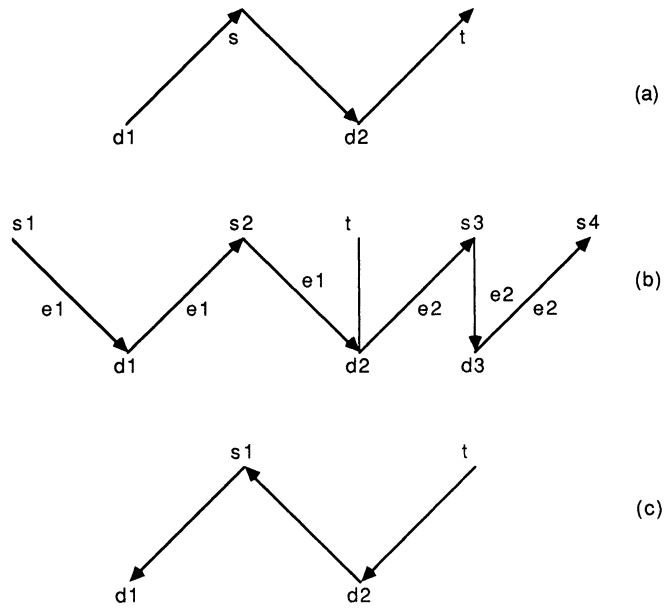


FIG. 1. *File transfer graph from the* RTT *problem.*

If t is a tight 2-teacher available during $\{t_2, t_3\}$ add nodes, s_1, d_1, d_2 as shown in Fig. 1(c). Thus t is available only during the second and last intervals for deadline $=3$.

We note that in all cases the maximum degree at any node is equal to 3. No extra construction is needed for tight 3-teachers since there are already 3 edges incident at such a node.

It therefore follows that G can be scheduled with deadline equal to 3 if and only if there is a valid timetable for I. Since the reduction may be performed in polynomial time, and the file transfer problem with forwarding is clearly in NP, the result follows.

2.2. Trees. We show that if there is no restriction on the maximum degree at a node and G is a tree, then FTPF is NP-complete. To do this we will reduce a scheduling problem in which tasks have discrete starting times to FTPF. We begin by introducing the scheduling problem and showing its NP-completeness.

Suppose we are given a set $\mathcal{T}=\{T_1, T_2, \cdots, T_n\}$ of n tasks and a single processor. Each task T_i requires execution time τ_i and has a set of possible starting times $s(T_i)=\{s_{i1}, s_{i2}, \cdots, s_{ik}\}$. Such scheduling problems for independent tasks have been studied in [12] and [13]. Suppose now that for each task T_i either $\tau_i=1$ or $\tau_i=2$ and T_i is a chain $b_{i1}<b_{i2}$ of unit length tasks b_{i1}, b_{i2}. We wish to know whether $\mathcal{T}$ can be scheduled on a single processor such that all precedences are satisfied. Proposition 1 states that the problem is NP-complete. We note that this result is of independent

interest since when no chains exist and all tasks have unit execution time the problem may be solved in polynomial time [13]. Adapting the notation of [12], we call our problem DSTP—discrete starting times with precedences problem.

PROPOSITION 1. *The* DSTP *problem is* NP*-complete in the strong sense even when all tasks have either unit execution time, and no precedence constraints, or are a chain* $b_{i1} < b_{i2}$ *of two unit execution time tasks.*

Proof. The DSTP problem is clearly in NP. We now reduce the 3-satisfiability problem (3SAT) to DSTP. We state 3SAT as follows (see [6, LO2, p. 259]).

Given a set U of p Boolean variables and a collection C of q clauses over U such that each clause in C has exactly three literals, is there a truth assignment for U such that each clause in C has at least one true literal?

Since 3-SAT remains strongly NP-complete even if for each $u \in U$ at most five clauses in C contain u or $\bar{u}$ [6, LO2, p. 259], we may assume this restriction holds.

Given any instance $\langle U, C\rangle$ of 3-SAT (with the above restriction) we construct a corresponding instance $\mathcal{T}(U, C)$ of the DSTP problem.

Let $U = \{u_1, u_2, \cdots, u_p\}$ and $C = \{c_1, c_2, \cdots, c_q\}$ and let $\bar{U} = \{\bar{u}_1, \bar{u}_2, \cdots, \bar{u}_p\}$. Let x_{l1}, x_{l2} and x_{l3} be the (distinct) literals in clause c_l.

For each $u_i \in U$, $1 \leqq i \leqq p$, we construct p_i unit execution time tasks u_{ij}, $j \in \{1, 2, \cdots, q\}$, where p_i is the number of times u_i or $\bar{u}_i$ occurs in C. By the assumption above, $1 \leqq p_i \leqq 5$. For each i let θ_i be a map

$$\theta_i : \{1, \cdots, p_i\} \to \{1, 2, \cdots, q\}$$

defined as $\theta_i(1) = r$ where c_r is the first clause of C in which u_i occurs, otherwise the first clause of C in which $\bar{u}_i$ occurs.

If $p_i > 1$, then $\theta_i(2), \cdots, \theta_i(p_i)$ are defined similarly, except only previously unassigned numbers (and clauses) are considered. Hence $u_{i\theta_i(1)}, \cdots, u_{i\theta_i(j)}$ correspond to clauses $c_{\theta_i(1)}, \cdots, c_{\theta_i(j)}$ in which u_i occurs and $u_{i\theta_i(j+1)}, \cdots, u_{i\theta_i(p_i)}$ correspond to clauses $c_{\theta_i(j+1)}, \cdots, c_{\theta_i(p_i)}$ in which $\bar{u}_i$ occurs for $1 \leqq j < p_i \leqq 5$.

We now create unit execution time tasks $a_{i\theta_i(1)}, \cdots, a_{i\theta_i(p_i)}$ and form the following chains:

$$a_{i\theta_i(j)} < u_{i\theta_i(j)} \quad \text{for } j \text{ odd, or}$$

$$u_{i\theta_i(j)} < a_{i\theta_i(j)} \quad \text{for } j \text{ even } 1 \leqq j \leqq p_i.$$

In addition we create p_i unit tasks $d_{i\theta_i(1)} \cdots d_{i\theta_i(p_i)}$. The set of possible starting times for each of the tasks created are defined as follows:

$$s(a_{i\theta_i(j)}) = \begin{cases} \left\{20(i-1) + \dfrac{j-1}{2},\ 20(i-1) + 7 + j\right\} & \text{if } j \text{ is odd,} \\ \left\{20(i-1) + 7 + j,\ 20(i-1) + 18 + \dfrac{j-2}{2}\right\} & \text{if } j \text{ is even,} \end{cases}$$

$$s(u_{i\theta_i(j)}) = \{20(i-1) + 2 + j,\ 20(i-1) + 12 + j\}.$$

We will write for each task T, $s_1(T)$ to denote the first possible starting time and $s_2(T)$ to denote the second, i.e., $s_1(T) < s_2(T)$. The start times for the $d_{i\theta_i(j)}$ depend on whether the literal is negated or nonnegated in $c_{\theta_i(j)}$. We consider the following possible cases. In each case "+" for $j = p_i$ is defined by $j + 1 = 1$.

(1) $u_i \in c_{\theta_i(j)}$, $u_i \in c_{\theta_i(j+1)}$

$$s(d_{i\theta_i(j)}) = \begin{cases} \{s_1(u_{i\theta_i(j)}),\ s_1(a_{i\theta_i(j+1)})\} & \text{if } j \text{ is even,} \\ \{s_2(u_{i\theta_i(j)}),\ s_2(a_{i\theta_i(j+1)})\} & \text{if } j \text{ is odd.} \end{cases}$$

(2) $u_i \in c_{\theta_i(j)}$, $\bar{u}_i \in c_{\theta_i(j+1)}$

$$s(d_{i\theta_i(j)}) = \begin{cases} \{s_1(u_{i\theta_i(j)}), s_2(u_{i\theta_i(j+1)})\} & \text{if } j \text{ is even,} \\ \{s_2(u_{i\theta_i(j)}), s_1(u_{i\theta_i(j+1)})\} & \text{if } j \text{ is odd.} \end{cases}$$

(3) $\bar{u}_i \in c_{\theta_i(j)}$, $\bar{u}_i \in c_{\theta_i(j+1)}$

$$s(d_{i\theta_i(j)}) = \begin{cases} \{s_2(u_{i\theta_i(j+1)}), s_2(a_{i\theta_i(j)})\} & \text{if } j \text{ is even,} \\ \{s_1(u_{i\theta_i(j+1)}), s_1(a_{i\theta_i(j)})\} & \text{if } j \text{ is odd.} \end{cases}$$

(4) $\bar{u}_i \in c_{\theta_i(j)}$, $u_i \in c_{\theta_i(j+1)}$

$$s(d_{i\theta_i(j)}) = \begin{cases} \{s_1(a_{i\theta_i(j+1)}), s_2(a_{i\theta_i(j)})\} & \text{if } j \text{ is even,} \\ \{s_1(a_{i\theta_i(j)}), s_2(a_{i\theta_i(j+1)})\} & \text{if } j \text{ is odd.} \end{cases}$$

We now show how the scheduling of the $u_{i\theta_i(j)}$ will be interpreted as the truth value of u_i. If $u_i \in c_{\theta_i(j)}$ then schedule $u_{i\theta_i(j)}$ at $s_2(u_{i\theta_i(j)})$ if j is odd, else at $s_1(u_{i\theta_i(j)})$ for j even. If $\bar{u}_i \in c_{\theta_i(j)}$ then schedule $u_{i\theta_i(j)}$ at s_1 for j odd, else at s_2 for j even.

This scheduling is consistent, i.e., scheduling any of the tasks $u_{i\theta_i(j)}$ as above according to a true value of u_i implies the other tasks are scheduled according to a true value of u_i. The details are given in Lemma A1 (of the Appendix).

Finally, we have q clause tasks c_l, $1 \leqq l \leqq q$, with unit execution times and three start times, one for each literal in c_l. We show how to construct the first start time; the construction for the next two start times is similar.

Suppose $u_i \in c_l$ or $\bar{u}_i \in c_l$; then $l = \theta_i(j)$ for some j, $1 \leqq j \leqq p_i$. We define

$$s_1(c_l) = \begin{cases} s_2(u_{il}) & \text{if } j \text{ is even,} \\ s_1(u_{il}) & \text{if } j \text{ is odd.} \end{cases}$$

Hence c_l may be scheduled during $\{20i, 20i+19\}$, the part of the schedule corresponding to u_i, if and only if u_{il} is scheduled according to u_i is true and $u_i \in c_l$, or u_{il} is scheduled according to $\bar{u}_i$ is true and $\bar{u}_i \in c_l$.

We claim $\langle U, C\rangle$ has a solution if and only if $\mathcal{T}(U, C)$ has a solution. Suppose that $\langle U, C\rangle$ has a solution. Let a truth assignment function $TA: U \cup \bar{U} \to \{true, false\}$ be such that each clause in C has at least one literal u with $TA(u) = true$. It is possible to construct a feasible schedule for all tasks in $\mathcal{T}(U, C)$ on a single processor as follows. For each $i \in \{1, 2, \cdots, p\}$ assign tasks $u_{i\theta_i(j)}$, $1 \leqq j \leqq p_i$, corresponding to u_i is true, otherwise schedule corresponding to u_i is false. For each clause c_l such that $u_i \in c_l$ and $TA(u_i) = true$, or $\bar{u}_i \in c_l$ and $TA(\bar{u}_i) = true$ then c_l can be scheduled at $s_2(u_{il})$ for j even or at $s_1(u_{il})$ for j odd where $l = \theta_i(j)$. Since all other tasks can always be scheduled whether the schedule corresponds to $TA(u_i) = true$ or $TA(\bar{u}_i) = true$ we conclude the schedule is feasible and is a solution for $\mathcal{T}(U, C)$.

Conversely, suppose $\mathcal{T}(U, C)$ has a solution. Let a truth assignment function $TA: U \cup \bar{U} \to \{true, false\}$ be defined as follows. If $u_{i\theta_i(j)}$, $1 \leqq j \leqq p_i$ are scheduled according to u_i is true, then $TA(u_i) = true$, otherwise $TA(u_i) = false$. By Lemma 1 the schedule of the $u_{i\theta_i(j)}$ is always consistent. If c_l is scheduled during $s_k(c_l)$ for some $k \in \{1, 2, 3\}$, then $TA(x_{lk}) = true$ where $x_{lk} = u_i$ or $\bar{u}_i$ for some u_i. This is true for each l, $1 \leqq l \leqq q$, therefore each clause in C has at least one literal u with $TA(u) = true$ and $\langle U, C\rangle$ has a solution.

We now consider the following example to illustrate the construction used in Proposition 1. Consider the instance of 3-SAT given by the following Boolean expression with three variables and four clauses:

$$(u_1 + \bar{u}_2 + u_3) \cdot (\bar{u}_1 + u_2 + u_3) \cdot (u_1 + \bar{u}_2 + \bar{u}_3) \cdot (\bar{u}_1 + \bar{u}_2 + \bar{u}_3)$$

and the corresponding task system

$$\mathcal{T}(U, C) = \{u_{11}, u_{12}, u_{13}, u_{14}, u_{21}, u_{22}, u_{23}, u_{24}, u_{31}, u_{32}, u_{33}, u_{34}, a_{11}, a_{12}, a_{13}, a_{14}, a_{21}, a_{22}, a_{23}, a_{24}, a_{31}, a_{32}, a_{33}, a_{34}, d_{11}, d_{12}, d_{13}, d_{14}, d_{21}, d_{22}, d_{23}, d_{24}, d_{31}, d_{32}, d_{33}, d_{34}, c_1, c_2, c_3, c_4\}.$$

Precedences and the map θ_i are described as follows:

$$\theta_1(1)=1, \quad \theta_1(2)=3, \quad \theta_1(3)=2, \quad \theta_1(4)=4,$$
$$a_{11}<u_{11}, \quad u_{13}<a_{13}, \quad a_{12}<u_{12}, \quad u_{14}<a_{14},$$
$$\theta_2(1)=2, \quad \theta_2(2)=1, \quad \theta_2(3)=3, \quad \theta_2(4)=4,$$
$$a_{22}<u_{22}, \quad u_{21}<a_{21}, \quad a_{23}<u_{23}, \quad u_{24}<a_{24},$$
$$\theta_3(1)=1, \quad \theta_3(2)=2, \quad \theta_3(3)=3, \quad \theta_3(4)=4,$$
$$a_{31}<u_{31}, \quad u_{32}<a_{32}, \quad a_{33}<u_{33}, \quad u_{34}<a_{34}.$$

Figure 2(a) illustrates the possible start times for each task, and Fig. 2(b) shows a feasible schedule for the truth assignment $u_1 = u_2 = \bar{u}_3 = \text{true}$.

2.2.1. NP-completeness of FTPF for trees. We now show the NP-completeness of FTPF for trees, by using a reduction from the DSTP scheduling problem.

THEOREM 2. *File Transfer Scheduling with forwarding is* NP*-complete for G a tree, all edge lengths equal, no multiple edges and* $\alpha(v) = 1$.

t=0	1	2	3	4	5	6	7	8	9	10	11	12	13
a11	a12		u11	u13	u12	u14		a11	a13	a12	a14		u11
d14	d12		C1	d13	C2	d12							d11

14	15	16	17	18	19	20	21	22	23	24	25	26	27
u13	u12	u14		a13	a14	a22	a23		u22	u21	u23	u24	
C3	d13	C4		d11	d14	d24	d23		C2	d22	C3	d23	

28	29	30	31	32	33	34	35	36	37	38	39	40	41
a22	a21	a23	a24		u22	u21	u23	u24		a21	a24	a31	a33
					d22	C1	d21	C4		d21	d24	d34	d33

42	43	44	45	46	47	48	49	50	51	52	53	54	55
	u31	u32	u33	u34		a31	a32	a33	a34		u31	u32	u33
	C1	d32	C3	d33							d31	C2	d32

56	57	58	59
u34		a32	a34
C4		d31	d34

FIG. 2(a). *The task system with possible execution intervals.*

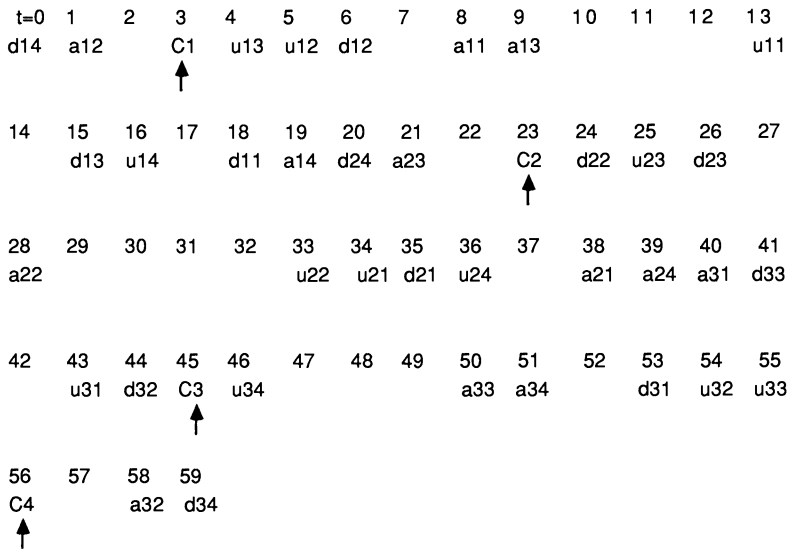


FIG. 2(b). *A feasible schedule for truth assignment* $u1 = u2 = \bar{u}3 = true$.

Proof. The FTPF problem clearly belongs to the class NP. We will now reduce to FTPF an instance of the DSTP problem in which tasks, u_i, either have unit execution time and no precedence constraints, or are a chain $b_{i1} < b_{i2}$ of two unit execution time tasks. In Proposition 1 we have shown that DSTP is NP-complete in the strong sense. Hence we may assume that the size of any integer occurring in DSTP is bounded by the length of the instance of our DSTP problem. By adding extra unit execution time tasks with no precedence constraints, if necessary, we may assume that a feasible schedule will have no idle time in our DSTP problem. Let d be the length of a feasible schedule in DSTP. We construct a file transfer problem with G a tree as follows.

Let G have root node v. For each unit execution time task u_i with no precedences, construct a child node w of v, which corresponds to a nonforwarded file transmission of length 1. By Lemma A2 (of the Appendix) we may construct a subtree with w as the root, in which transmissions between v and w may only occur at the set of possible start times for u_i, $s(u_i)$.

If $b_{i1} < b_{i2}$ is a chain of unit execution time tasks, we construct child nodes w_{i1} and w_{i2} of G corresponding to a forwarded file to be sent from w_{i1} to v and from v to w_{i2}. Again by Lemma A2, we may assume that possible transmissions between w_{i1} and v can occur only during times in $s(b_{i1})$ and between v and w_{i2} only during times in $s(b_{i2})$. By our assumption that follows from the strong NP-completeness of DSTP, all constructions may be carried out in polynomial time. Since FTPF can be scheduled with deadline d if and only if DSTP has a feasible schedule, we conclude that FTPF is NP-complete.

2.2.2. Algorithms for trees. Throughout this section we shall suppose that G is a tree with equal edge lengths, no multiple edges, and $\alpha(v) = 1$. We begin by examining the special case of a tree in which the only node sending or receiving more than one file is the root node.

LEMMA 1. *Suppose the root is the only node of a tree in which more than one file is sent/received (apart from forwarded files); then the earliest deadline first algorithm is*

optimal for scheduling files at the root to meet an overall deadline d. This algorithm runs in time $O(r \log r)$ where r is the degree of the root.

Proof. Let v be the root of the tree G. All branches into v are by hypothesis single paths. We reduce this scheduling problem to the problem of scheduling tasks on a single processor with release times and deadlines [7]. Suppose a branch into v represents an incoming file as shown in Fig. 3(a). If $j \geqq 2$, then the file may be scheduled to leave v_0 at time $t=0$ and hence arrives at v_{j-1} at time $j-1$. Then the scheduling at v corresponds to scheduling a unit execution time task on a single processor with release time $j-1$ and deadline d. If $j=1$ then $v_{j-1}=v_0$ is adjacent to v and the release time is zero at v_0.

Suppose a branch represents an out-going file as shown in Fig. 3(b), then the scheduling at v corresponds to scheduling a unit execution time task with release time zero and deadline $d-(j-1)$.

Suppose a file is forwarded at v and is represented by the paths as shown in Fig. 3(c). Then the scheduling at v may be represented by two unit execution time tasks b_1, b_2 with $b_1 < b_2$, i.e., b_2 starts only after b_1 has completed. We let b_1 have release time $j-1$ and deadline $d-k$, and b_2 have release time j and deadline $d-(k-1)$. As observed in the introduction to [7], Lemma 2 of [7] shows that for one processor, the partial order may be ignored and then the result follows directly from algorithms for scheduling independent tasks with release times and deadlines (see [11]).

We note that for discrete starting times the partial order may no longer be ignored (see the previous section).

We now introduce some definitions for investigating polynomial time algorithms for general trees.

DEFINITION. Let G be a tree. If v is a node of G then v is a *branch* node if:

(1) v is a leaf node;

(2) There is more than one file sent, received or forwarded through v.

Note. The only nodes in which only one file is sent or received (not forwarded) are the leaf nodes.

We shall now distinguish between three possible types of branch nodes v in the tree.

Type I. There is a path representing a file to be forwarded down the tree from its nearest branch ancestor v' to v and the path from v' to v is of length greater than or equal to 2. The file may or may not be forwarded down the tree.

Type II. Same as type I except the file is forwarded up the tree from v to v'. The file may or may not have been forwarded to v from below in the tree.

Type III. The parent v' of v is a branch node and there is a file to be transmitted between v and v'. This file may or may not be forwarded up/down the tree.

All other nodes are forwarding nodes through which only one file passes.

Suppose G is a tree in which all branch nodes are of type I or type II. Given a deadline d, the following algorithm produces a schedule of length d (if possible). We

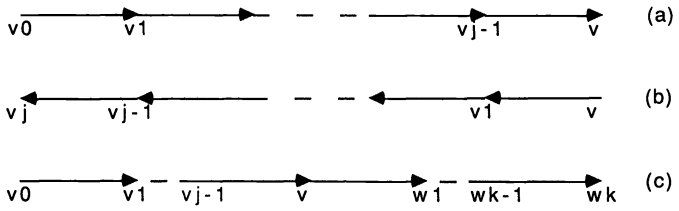


FIG. 3. *Branches into the root v of a tree.*

assume without loss of generality that $d < |E|$; otherwise, if $d \geqq |E|$ we can schedule all edges sequentially. We note that by hypothesis there is always an intermediate forwarding node between branch nodes, so once a file is ready it may be scheduled at any time by the branch node.

ALGORITHM A.

(1) Schedule all leaves of the tree that are of type II, by scheduling the file f to be sent from the leaf at $t = 0$.

(2) Choose a node v all of whose descendant branch nodes v'' have been scheduled. As in Lemma 1, if v'' sends a file, f, to v, calculate the ready time of f at v. Similarly, if v'' receives a file, f, from v we can calculate the deadline for sending f from v.

(a) TYPE I NODE.
Schedule the file f from v' as late as possible. For each t, $0 \leqq t \leqq d-1$ (starting with $t = d-1$), assume f has ready time t and deadline $t+1$. If f is forwarded, adjust the ready time for the forwarded step to be $t+1$. Then schedule v by earliest deadline first algorithm as in Lemma 1. If a schedule is found stop, else try next value of t.

(b) TYPE II NODE.
Schedule the file f from v to v' as early as possible. For each t, $0 \leqq t \leqq d-1$ (starting with $t = 0$), assume f has ready time t and deadline $t+1$. If f is forwarded at v, and the ready time at v for the first transmission is s, adjust the deadline at v for the first transmission to be t and only consider values of t for $s+1 \leqq t \leqq d-1$. Schedule v by the earliest deadline first algorithm as in Lemma 1. If a schedule is found stop, else try next value of t.

(3) Repeat step (2) until all branch nodes have been scheduled. If at any stage a node may not be scheduled then no overall schedule is possible. Finally, nonbranch nodes can now easily be scheduled. If the node is adjacent to a branch node its schedule is already determined. Otherwise, schedule the (forwarded) file through the node at its ready time.

THEOREM 3. *If G is a tree in which all branch nodes are of type* I *or type* II, *then G is scheduled with deadline d by Algorithm* A *in time $O(n|E|^2 \cdot \log |E|)$, where n denotes the number of branch nodes in G. If Algorithm* A *fails then no schedule is possible.*

Proof. Consider a branch node v. If v is a leaf of type II then Algorithm A schedules the file at v to leave at $t = 0$. This is the best possible time. If v is a branch node of type I the file leaving v' for v is sent as late as possible to v, assuming v can be scheduled to meet the deadline d. This ensures the deadline for f at v' is as late as possible in any feasible schedule. Similarly, if v is a branch node of type II, the file leaving v for v' is sent as early as possible from v, assuming v can be scheduled to meet the deadline d. This ensures the ready time for f at v' is as early as possible in any feasible schedule. Hence if Algorithm A fails no schedule is possible.

Since step (2) may be performed in time no more than $O(|E|^2 \cdot \log |E|)$ this gives the worst-case time bound as stated (since $d < |E|$).

Note. To obtain an optimal schedule run a binary search on deadline d. Thus an optimal schedule can be obtained in time $O(n|E|^2 \cdot \log^2 |E|)$.

We conclude that if branch nodes are not permitted to be adjacent to each other, then the scheduling problem may be solved in polynomial time. As is evident from Theorem 2, once branch nodes are permitted to be adjacent to each other the problem becomes NP-complete. However, if we bound the maximum degree allowed at a branch node, the problem once more may be solved in polynomial time.

Enumerating all possible schedules at a branch node v in the general case may be done in time $O(|E|^r)$ where $r = \text{degree}(v)$. (As before we assume $d < |E|$.) Hence if we bound the degree of each (branch) node, we obtain the following algorithm.

ALGORITHM B.

(1) Schedule all leaves of the tree which are of type II by scheduling the file f to be sent from the leaf at $t = 0$.

(2) Choose a node v all of whose descendant branch nodes v'' have been scheduled. If v'' is a descendant branch node of type III then v'' is adjacent to v, and we may assume by a straightforward induction argument that we have a set $S_{v''} \subseteq \{0, 1, \cdots, d-1\}$ of available times for a transmission between v and v''. If v'' is a descendant branch node of type I or II, then the node v_j adjacent to v on the (unique) path from v to v'' has a ready time and a deadline for a transmission between v and v'' (as in Algorithm A).

(a) TYPE I NODE.
Search for a feasible schedule of v such that file f from v' to v is sent as late as possible (cf. Algorithm A).

(b) TYPE II NODE.
Search for a feasible schedule of v such that file f from v to v' is sent as early as possible (cf. Algorithm A).

(c) TYPE III NODE.
In this case v and v' are adjacent. For each $t \in \{0, 1, \cdots, d-1\}$ schedule the file between v and v' to be sent at time t. If a feasible schedule for v is found, then $t \in S_v \subseteq \{0, 1, \cdots, d-1\}$; otherwise, $t \notin S_v$.

(3) Repeat step (2) until all branch nodes have been scheduled. If at any stage a node may not be scheduled then no overall schedule is possible. Nonbranch nodes may be scheduled as in Algorithm A.

THEOREM 4. *If G is a tree such that the maximum degree of a node is bounded by some integer R, then G is scheduled with deadline d by Algorithm* B *in time*

$$O(nR|E|^R \cdot \log|E|)$$

where n is the number of branch nodes in G. If Algorithm B *fails then no schedule is possible.*

Proof. Since Algorithm B is essentially an exhaustive search of all possible schedules at each branch node, it is clear that if Algorithm B fails then no schedule is possible.

At each branch node, each possible schedule may be checked for feasibility in time no more than $R \cdot \log d$. Since there are $O(|E|^R)$ schedules to check, the time bound for Algorithm B is as stated.

COROLLARY. *An optimal schedule may be obtained in time $O(nR|E|^R \cdot \log^2|E|)$.*

2.3. Algorithms for paths and cycles. In this section we examine the special case of paths and cycles, and obtain algorithms with better worst-case time bounds. We begin by classifying branch nodes into three distinct types.

DEFINITION. Let v be any branch node of G that has v' and v'' as its nearest branch nodes. Suppose file f_1 is to be transmitted between v' and v and f_2 is to be transmitted between v and v''. Then v is said to be as follows.

Type A. If f_1 is sent from v to v' and f_2 is sent from v to v''.

Type B. If f_1 is sent from v' to v and f_2 is sent from v'' to v.

Type C. If f_1 is sent from v' to v and f_2 is sent from v to v'' or f_1 is sent from v to v' and f_2 sent from v'' to v.

We now examine how to schedule type A nodes in the case where files do not have the same number of forwarding steps.

LEMMA 2. *Let G be a path or cycle. If v is any node of type* A, *then in any schedule it is optimal to transmit the file with the maximum number of forwarding steps first in the case where the files do not have the same number of steps.*

Proof. Let t_0 be the time taken to finish transmitting all files from v if scheduled as above, and let t_1 be the time taken to schedule the shortest number of steps file first. Let f_1 and f_2 be as described in the definition. Without loss of generality suppose f_1 has l_1 steps, and that f_2 has l_2 steps where $l_1 > l_2$.

If f_1 is sent first, then f_2 is sent at $t=1$ in the case $l_2 > 1$. If v' is busy at time l_2 in either case, then f_2 arrives at v' in time no more than l_1+1. If v'' is busy at $t=l_1-1$, then f_1 arrives at v'' in time no more than l_1+1. Hence $t_0 \leqq l_1+1$.

Suppose now that f_2 is sent first. Then f_1 can arrive at v'' no earlier than l_1+1. Hence $t_1 \geqq l_1+1$, which implies that $t_0 \leqq t_1$. Therefore this scheduling is optimal.

We now examine how to schedule files that must be transmitted to/from type B and type C nodes.

LEMMA 3. *Let G be a path or cycle. Suppose v is not adjacent to two branch nodes.*

(1) *Suppose v is any type* C *node. If the out-going file must be forwarded it is optimal to send it at $t=0$. Otherwise, if there is a tie, schedule arbitrarily.*

(2) *If v is any type* B *node it is optimal to schedule transmissions at* B *as soon as files are ready. These ready times are determined by the scheduling of nodes of type* A *and type* C. *If there is a tie then schedule arbitrarily at v.*

Proof. (1) If the out-going file must be forwarded and is the only file ready at $t=0$ it is clearly optimal to send it at that time. Otherwise, if v is adjacent to a branch node, then v will be finished under any schedule in three time units. Since the out-going file takes at least three time units if sent at time $t=1$, scheduling the out-going file at $t=0$ is clearly optimal.

If v is adjacent to a branch node and the out-going file is to be transmitted to that branch node, if the node is not available at $t=0$ then it is available at $t \geqq 1$. So we may schedule v arbitrarily in the case of a tie.

(2) By hypothesis v is not adjacent to two branch nodes. Therefore if there is a tie at time t, then $t \geqq 1$. Hence if B is not adjacent to a branch node we may schedule files arbitrarily at v since no node is busy with another file at $t+1$. On the other hand, if v is adjacent to a branch node, the branch node is not busy at $t \geqq 1$, since it must have transmitted its other file at $t=0$. Again we can schedule arbitrarily.

DEFINITION. Let G be a file transfer graph. Then makespan (G) is the smallest possible overall finishing time in any feasible schedule.

In any graph G let L_1 equal the maximum number of forwarding steps of any file. Then makespan $(G) = L_1$, L_1+1 or L_1+2. (Delay for any file being sent is at most one time unit at initial and terminal nodes.)

The type A nodes that have files of an equal number of steps j have smallest finishing time $j+1$ and maximum finishing time $j+2$. Let L_2 be the maximum number of forwarding steps of any such type A node in G. We need only consider these type A nodes when $j=L_2$ and $L_2+1=L_1$, i.e., $L_2=L_1-1$. All other such nodes may be scheduled arbitrarily. Let L_3 equal the maximum number of steps of any file in a type A or type B node in which both files have the same number of steps. We obtain the following result for paths and cycles.

THEOREM 5. *Let G be a path or a cycle. Then an optimal schedule for G may be obtained in time $O(|E|)$ and* makespan (G) *is given by the following*:

(1) *Suppose $L_3 = L_1$. Then* makespan $(G) = L_1+1$.

(2) *Suppose* $L_3 < L_1$ *and* $L_2 = L_1 - 1$. *Then* makespan $(G) = L_1 + 1$ *in the case where* G *has a path* $u_1, u_0, v_0, v_1, \cdots, v_{n-1}, u_{n-1}, u_n$ *as classified in Fig.* 4. *Otherwise,* makespan $(G) = L_1$.

(3) *Suppose* $L_3 < L_1$ *and* $L_2 < L_1 - 1$. *Then* makespan $(G) = L_1$.

Proof. (1) We may assume that $L_1 > 1$; otherwise, if $L_1 = 1$, the result follows from [4]. Suppose that $L_2 < L_3$. Then all type A nodes will have their files finished with time at most $L_2 + 2 = L_1 + 1$. We now consider the type B node v as shown in Fig. 5(a).

By hypothesis, v'' and v' must be type C nodes in the case where their other file takes L_3 steps. Otherwise, they are type A or type C nodes in which the other file takes less than L_3 steps. By Lemmas 2 and 3, an optimal schedule will send f_0 and f_1 from v'' and v' at $t = 0$. Hence v is finished by $L_1 + 1$. So makespan $(G) \geqq L_1 + 1$. Since all files of length L_1 will be sent at $t = 0$ by Lemmas 2 and 3, we obtain makespan $(G) = L_1 + 1$ as required.

Suppose that $L_1 = L_2 = L_3$. As in the previous case it is clear that makespan $(G) \geqq L_1 + 1$. Any file with L_1 steps from a type C node will finish in time at most $L_1 + 1$ by Lemma 3. We therefore consider files sent from type A nodes. If the closest branch nodes are type C nodes the result is clear. We therefore consider type A nodes that have type B nodes as their closest branch nodes.

Consider a maximal length path, from a node v_0 to a node v_{n-1} in which there are n vertices $v_0, v_1, \cdots, v_{n-1}$ of alternating type A and type B nodes each of which

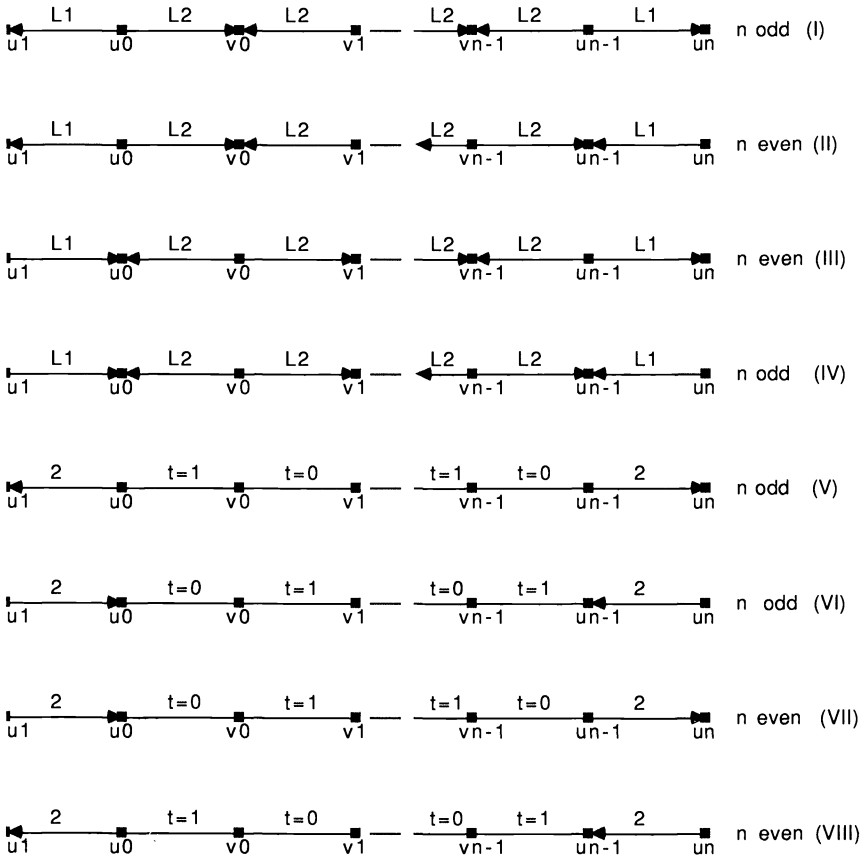


FIG. 4. *Classification of paths of G.*

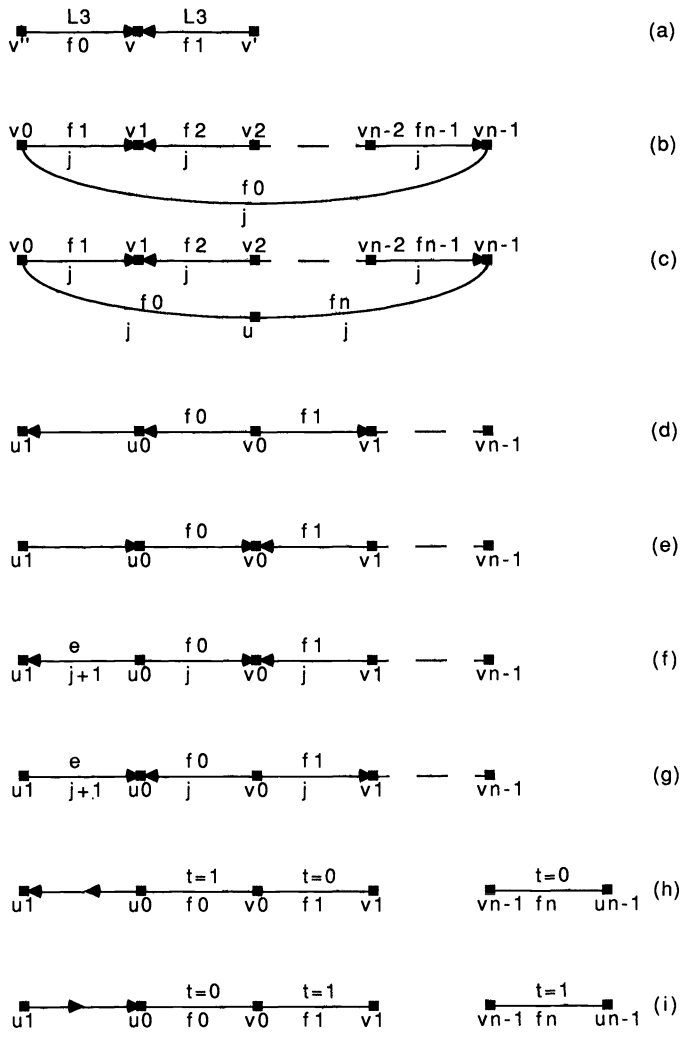


FIG. 5. *Classification of some node types with schedules.*

have files of equal number of steps, j, where $j = L_1$, and such that v_i and v_{i-1} are closest branch nodes, $0 \leqq i < n-1$.

Suppose that G is a cycle and v_0 and v_{n-1} are closest branch nodes. Then, without loss of generality we must have that v_0 is type A, v_{n-1} is type B and n is even (see Fig. 5(b)). If f_i is scheduled at $t=0$ from v_{i-1}, $1 \leqq i \leqq n-1$ and i is odd, then f_k is scheduled at $t=1$ from v_k, $0 \leqq k \leqq n-2$ and k is even. Maximum finishing time for any node is then L_1+1.

Suppose now that v_0 and v_{n-1} are not closest branch nodes and let u_0 be the closest branch node to v_0, u_{n-1} the closest branch node to v_{n-1}. If $u = u_0 = u_{n-1}$, and G is a cycle then by maximality of n we have without loss of generality that u is of type C, v_0 is of type A, v_{n-1} is of type B, and n is even (see Fig. 5(c)). If f_i is scheduled at $t=0$ from v_{i-1}, $1 \leqq i \leqq n-1$ and i is odd, then f_k is scheduled at $t=1$ from v_k, $0 \leqq k \leqq n-2$ and k is even, and f_n is scheduled from u at $t=0$. The maximum finishing time for any node is then L_1+1. Hence we may suppose that u_0 and u_{n-1} are distinct.

By maximality of n, either u_i, $i=0$, $n-1$ is of type C or u_1 is of type A or type B and has its other file with k steps where $k<j$.

Case (i). u_0 is of type C. Let u_1 be the closest branch node to u_0. We have either the situation of Fig. 5(d) or 5(e). Consider first the situation of Fig. 5(d). By Lemma 3, u_0 is free at $t=L_1=j$. Send f_i at $t=1$ from v_i, and f_{i+1} at $t=0$ from v_i for i even. Then the maximum finishing time is L_1+1. In the situation of Fig. 5(e), by Lemma 3, f_0 is sent at $t=0$ from u_0. Hence v_0 is free at $t=j$. Send f_i at $t=1$ from v_i, and f_{i+1} at $t=0$ from v_i for i odd. Then maximum finishing time is L_1+1.

Case (ii). u_0 is of type A or type B. Then the file e between u_0 and u_1 has k steps where $k<j$. Consider the following situations:

(a) v_0 is of type A. Then u_0 is of type B. The latest arrival time at u_0 for file e is $k+1$, if there is no delay at u_0. Hence assume that u_0 is free at $t=j$ and schedule as in Case (i) of Fig. 5(d). Hence maximum finishing time is L_1+1.

(b) v_0 is of type B. Then u_0 is of type A. By Lemma 2, f_0 is sent at $t=0$ from u_0. Hence f_0 is finished at time $t=j$ at v_0 if there is no delay. We may assume that v_0 is free at $t=j$ and schedule as in Case (i) of Fig. 5(e). Hence the maximum finishing time equals L_1+1.

We conclude that in all the cases in (1) that makespan $(G)=L_1+1$.

(2) Suppose $L_3<L_1$ and $L_2=L_1-1$. As in case (1) the only difficulty will be maximal paths with type A and type B nodes that are closest branch nodes.

Assume that $L_2>1$. Consider the maximum length path, with $v_0, v_1, \cdots, v_{n-1}$ alternating type A and type B closest branch nodes each of which have files of equal number of steps j, where $j=L_1-1$. Let u_0 be the closest branch node to v_0, and u_{n-1} be the closest branch node to v_{n-1}. By hypothesis, u_0 and u_{n-1} are distinct.

Case (i). u_0 is of type C. In all cases send the out-going file from u_0 at $t=0$. Then schedule the same as (1) Case (i). Hence the maximum finishing time is L_1.

Case (ii). u_0 is of type A or of type B, and the file e_0 between u_0 and u_1 has k steps where $k<j$. Then schedule as in (1) Case (ii). The maximum finishing time is L_1.

Case (iii). u_0 is of type A or of type B and the file e between u_0 and u_1 has $j+1$ steps. Consider the situation as shown in Fig. 5(f). Then send the file f_0 at $t=1$ from u_0, f_{i+1} at $t=1$ from v_i, for i odd and f_i at $t=0$ from v_i for i odd. Suppose that v_{n-1} is a type B node. Then n is odd and f_{n-1} is scheduled at $t=1$ from v_{n-2}. The maximum finishing time is L_1 if f_n is sent at $t=0$ otherwise the maximum finishing time is L_1+1 and u_n, u_{n-1} have a file with $j+1$ steps. We therefore have the situation of Fig. 4(I). Suppose v_{n-1} is a type A node. Then n is even and f_n is sent at $t=1$. The maximum finishing time is then L_1 unless we have the situation of Fig. 4(II) in which case it is L_1+1. Consider now the situation as shown in Fig. 5(g). Then e is sent at $t=0$. Hence send f_0 at $t=0$ from v_0, f_i at $t=0$ from v_i for i even and f_{i+1} at $t=1$ from v_i for i odd. Suppose that v_{n-1} is a type B node. By symmetry the schedule has maximum finishing time L_1 unless there is the situation of Fig. 4(I) or (III) in which case it is L_1+1. Suppose v_{n-1} is a type A node. Then the maximum finishing time is L_1 unless there is the situation of Fig. 4(IV) in which case it is L_1+1.

Now consider the case where $L_2=1$ and $L_1=2$. By maximality of n assume the file between u_1 and u_0, u_{n-1} and u_n has two steps. If n is odd and we have the situation of Fig. 5(h) we can schedule f_i at $t=1$ for i even and have the maximum finishing time is L_1 unless there is the situation of Fig. 4(V). If we have the situation of Fig. 5(i) then we can schedule f_i at $t=0$ for i even and the maximum finishing time is L_1 unless Fig. 4(VI) holds. In the case of n even a similar argument shows that the maximum finishing time is L_1 unless Fig. 4(VII) or (VIII) holds. Since it is clear that all schedules are optimal the statement of the theorem is true for (2).

(3) This follows directly from the discussion before the theorem and from Lemmas 2 and 3.

Finally, we note that we have shown that Algorithm C (below) is optimal. Since Algorithm C runs in time $O(|E|)$ the theorem follows.

ALGORITHM C.

(1) Calculate L_3, L_2, L_1. If $L_1 = 1$, then schedule by [4]; otherwise $L_1 > 1$.

(2) Schedule type A nodes that have files of a different number of steps according to Lemma 2.

(3) Schedule type B and type C nodes that are not adjacent to two branch nodes according to Lemma 3.

(4) If $L_1 = L_3 > L_2$, schedule all other nodes arbitrarily. Otherwise, if $L_2 > 1$ and $v_0, v_1, \cdots, v_{n-1}$ is a maximal path of alternating type A and type B closest branch nodes with both files having L_2 steps, schedule as follows:

(a) v_0 is of type A. Let u_0 be the closest branch node to v_0. If there is no file of length L_1 into u_0 then schedule f_i at $t = 1$ from v_i (see Fig. 6(a)), otherwise schedule f_i at $t = 0$ from v_i for i even.

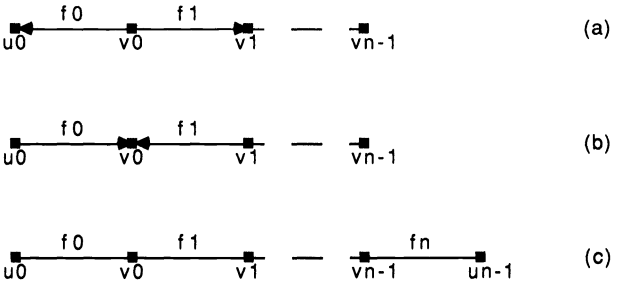


FIG. 6. *Scheduling of some node types.*

(b) v_0 is of type B. Let u_0 be the closest branch node to v_0. If there is no file of length L_1 from u_0 to u_1 then schedule f_i at $t = 0$ from v_{i+1} (see Fig. 6(b)), otherwise schedule at $t = 1$ where i is even.

If $L_2 = 1$ let $u_0, v_0, \cdots, v_{n-1}, u_{n-1}$ be a maximal path of branch nodes such that each node is adjacent to the next branch node as shown in Fig. 6(c). Schedule f_i for i even at $t = 0$ if there is no path of length 2 from u_0 to u_1, otherwise schedule at $t = 1$.

(5) Schedule any remaining nodes arbitrarily.

3. Complexity results for arbitrary edge lengths.

3.1. NP-completeness results. It has been shown in [4] that except for certain special cases of paths and cycles, the file transfer problem without forwarding is NP-complete, when arbitrary edge lengths are allowed. We now show that in all these cases where multiple edges are allowed, the file transfer problem with forwarding becomes NP-complete. Since all other cases have been covered in [4], except paths and even cycles, the next theorem proves NP-completeness in these new cases.

THEOREM 6. FTPF *is* NP*-complete for paths and even cycles in which multiple edges are allowed and* $\alpha(v) = 1$ *for all* $v \in V$.

Proof. We shall reduce the NP-complete problem PARTITION to the two cases of FTPF as stated in the theorem. We state the PARTITION problem:

> Given a sequence $A = (a_1, a_2, \cdots, a_n)$ of positive integers does there exist a subset $A' \subseteq A$ such that $\sum_{a \in A'} a = \frac{1}{2} \sum_{i=1}^{n} a_i$.

Given an instance of PARTITION we show the corresponding graph for the case of paths in Fig. 7(a) where $B = \frac{1}{2}\sum_{i=1}^{n} a_i$ and deadline is $5B$. The case of even cycles is shown in Fig. 7(b) and deadline is also $5B$.

Each graph can be constructed in polynomial time so the proof is complete if we show schedules exist with deadline $5B$ if and only if the desired partition exists.

First suppose there is a partition $A_1 \cup A_2 = A$ such that $\sum_{a_i \in A_1} a_i = \sum_{a_i \in A_2} a_i = B$. A feasible schedule is obtained as follows. At time $t = 0$ transmit f_0 from v_0, and at time $t = 2B$ transmit to v_1. The B files $e_i \in A_1$ are transmitted sequentially between v_1 and v_2 starting at $t = 0$. At time $t = B$ transmit f_1 from v_2, and at time $t = 3B$ send f_1 to v_3. Finally, the B files e_i, $e_i \in A_2$ are transmitted sequentially between v_1 and v_2 starting at $t = 4B$. In the case of an even cycle send f_2 between v_0 and v_3 starting at time $t = 2B$. Hence the desired schedule exists.

Next suppose there exists a schedule with finishing time at most $5B$. In each case we obtain:

(1) Transmission of f_0 to v_1 and transmission of f_1 from v_2 must overlap for exactly B units of time (see [4] proof of Theorem 9) in order for the deadline to hold.

(2) f_0 must be sent to v_1 at its earliest possible time of $t = 2B$ and f_1 must be sent from v_1 at the latest possible time of $t = B$; otherwise (1) does not hold.

(3) By precedence constraints f_0 must be transmitted from v_0 at $t = 0$ and f_1 must be transmitted to v_3 at $t = 3B$.

(4) v_0 and v_2 are available simultaneously only during the intervals $[0, B]$ and $[4B, 5B]$. Hence the files in A must be sent during these times. Since their lengths total exactly $2B$, they completely occupy these two regions. Since the lengths of the subsets of A in each region must sum to exactly B, we obtain the desired partition.

3.2. Algorithms for paths and cycles. In the previous section we have shown that for multiple edges FTPF is NP-complete for paths and even cycles. For single edges, the problem has been shown in [4] to be NP-complete for trees. We therefore examine the single edge situation for paths and cycles, noting that in [4] polynomial time algorithms have been obtained in these cases when forwarding is not permitted.

The next two lemmas examine how to obtain an optimal schedule at a node when it is not adjacent to other branch nodes, and there is at most one out-going file from the node.

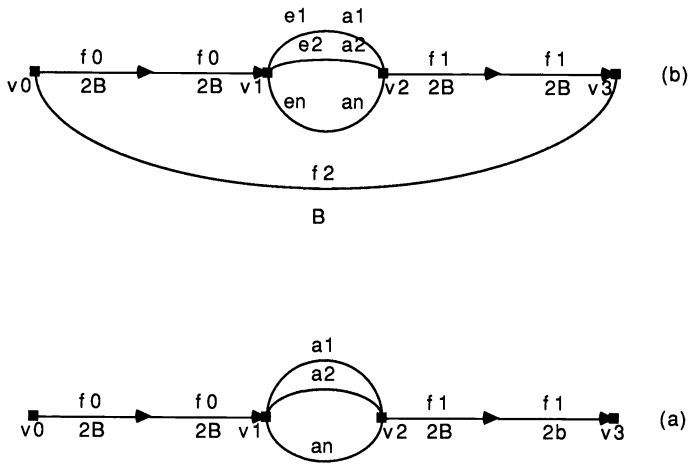


FIG. 7. *Graphs corresponding to the* PARTITION *problem.*

LEMMA 4. *In any schedule, given a path with branch nodes v_0, v_1, v_2 not adjacent to each other, as in Fig.* 8(a), *it is always optimal to schedule e_1 before e_0 at v_1.*

Proof. Let S be any schedule. Let d_0 be delay at v_0 before e_0 is sent in S. Let j be the number of steps of e_0. By hypothesis, $j \geqq 2$. Since e_1 is ready at v_1 at time $t=0$, it is clear that the finishing time of e_1 at v_2 is optimal when e_1 is sent at $t=0$.

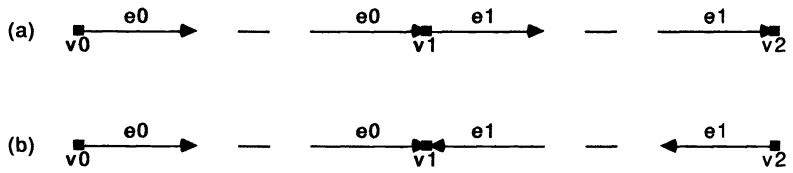


FIG. 8. *Scheduling of nonadjacent branch nodes.*

The time e_0 is sent from v_0 is fixed so it remains to prove that v_1 finishes optimally when e_1 is sent first from v_1. If e_1 waits for e_0 then v_1 takes time $d_0+ja_0+a_1$ where a_i denotes the length of e_i for $i=1,2$. If e_1 does not wait for e_0 then v_1 takes time t where

$$t=\begin{cases} a_1+a_0 & \text{if } d_0+(j-1)a_0 \leqq a_1, \\ d_0+ja_0 & \text{if } d_0+(j-1)a_0 > a_1. \end{cases}$$

Hence the proposed schedule is optimal.

LEMMA 5. *In any schedule, given a path as in Fig.* 8(b) *with branch nodes v_0, v_1, and v_2 not adjacent to each other, it is always optimal to schedule the first ready file at v_1.*

Proof. Let S be any schedule. Let j be the number of steps of e_0, k be the number of steps of e_1, d_0 be the delay at v_0 of file e_0 in S, and d_1 be the delay at v_1 of file e_1 in S. Then e_0 is ready at v_1 at time $d_0+(j-1)a_0$, and e_1 is ready at v_1 at time $d_1+(k-1)a_1$.

Suppose that $d_0+(j-1)a_0 < d_1+(k-1)a_1$. Then if e_0 waits until e_1 is finished at v_1 the finishing time at v_1 is $d_1+ka_1+a_0$. If e_0 is scheduled at v_1 at time $d_0+(j-1)a_0$, then the finishing time t at v_1 is given by

$$t=\begin{cases} d_0+ja_0+a_1 & \text{if } d_1+(k-1)a_1 \leqq d_0+ja_0, \\ d_1+ka_1 & \text{if } d_1+(k-1)a_1 > d_0+ja_0. \end{cases}$$

In the first case,

$$d_0+ja_0+a_1 = d_0+(j-1)a_0+a_0+a_1 < d_1+(k-1)a_1+a_1+a_0 = d_1+ka_1+a_0.$$

In the second case,

$$d_1+ka_1 < d_1+ka_1+a_0.$$

Hence the proposed schedule is optimal. By symmetry the result is true if $d_1+(k-1)a_1 < d_0+(j-1)a_0$.

We conclude that we must decide how to schedule the branch nodes v_1 as shown in Fig. 9(a). We note that it is sufficient to schedule these nodes so that the nodes u_1 of Fig. 9(b) meet the deadline d.

Unfortunately, Fig. 10 gives an example where it is not optimal to send the longest number of steps file first. An optimal schedule sends f_0 and f_2 at $t=0$, f_1 and f_3 at $t=2$ to give makespan $=4$ (optimal). If f_0 and f_3 are sent at $t=0$ we obtain makespan $=5$. Hence we are unable to prove a result analogous to Lemma 2, for nodes that have two out-going files.

We note however, that the finishing time at each node is one of at most five possible values, each depending on the fact that just two particular files were started

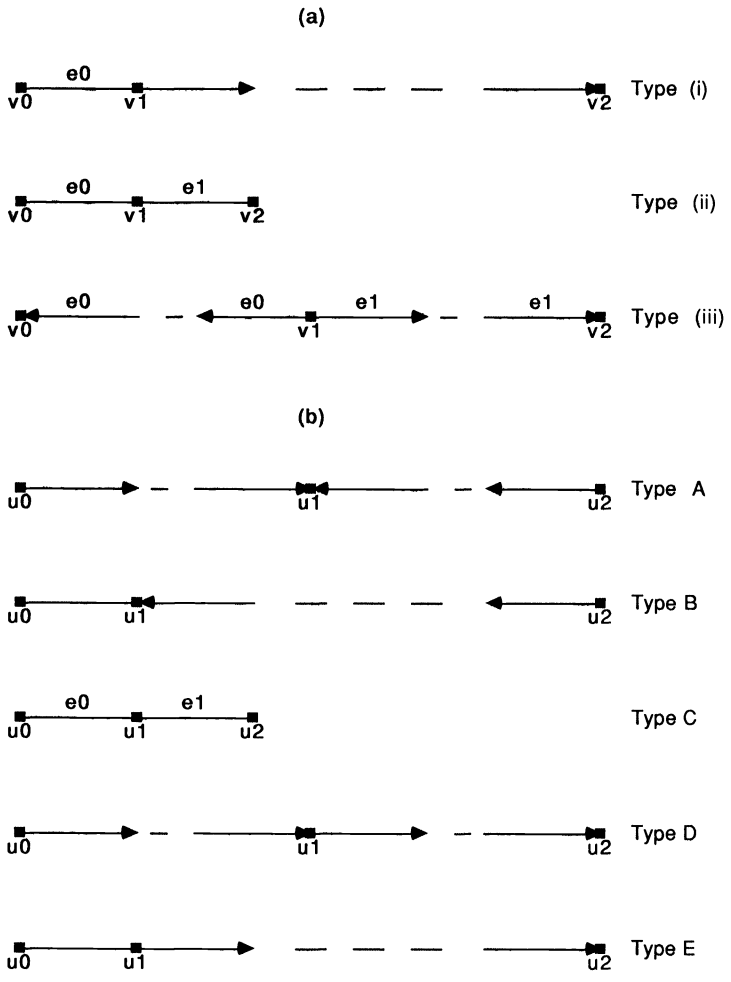


FIG. 9. *Critical nodes to be scheduled.*

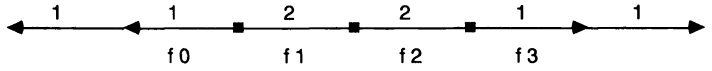


FIG. 10. *Example to show that sending the longest steps file first is not optimal.*

at $t = 0$. (The rest of the schedule follows by Lemmas 4 and 5.) Once we have established this fact (Lemma 6) we will reduce the file scheduling problem to the 2-satisfiability problem (2-SAT) giving a polynomial time solution. Note that 2-SAT is analogous to 3-SAT, defined in § 2, except that each clause has at most two literals.

The nodes for which we need to establish possible starting times, are classified and listed in Fig. 11. These are essentially all nodes of types A, B, C, D, E in all possible contexts that they might appear in G.

LEMMA 6. *Let G be a path or cycle. For each branch node that is either a terminal node for some file, or adjacent to another branch node, there are at most five possible finishing times. Each finishing time depends on the fact that at most two particular files are sent at $t = 0$.*

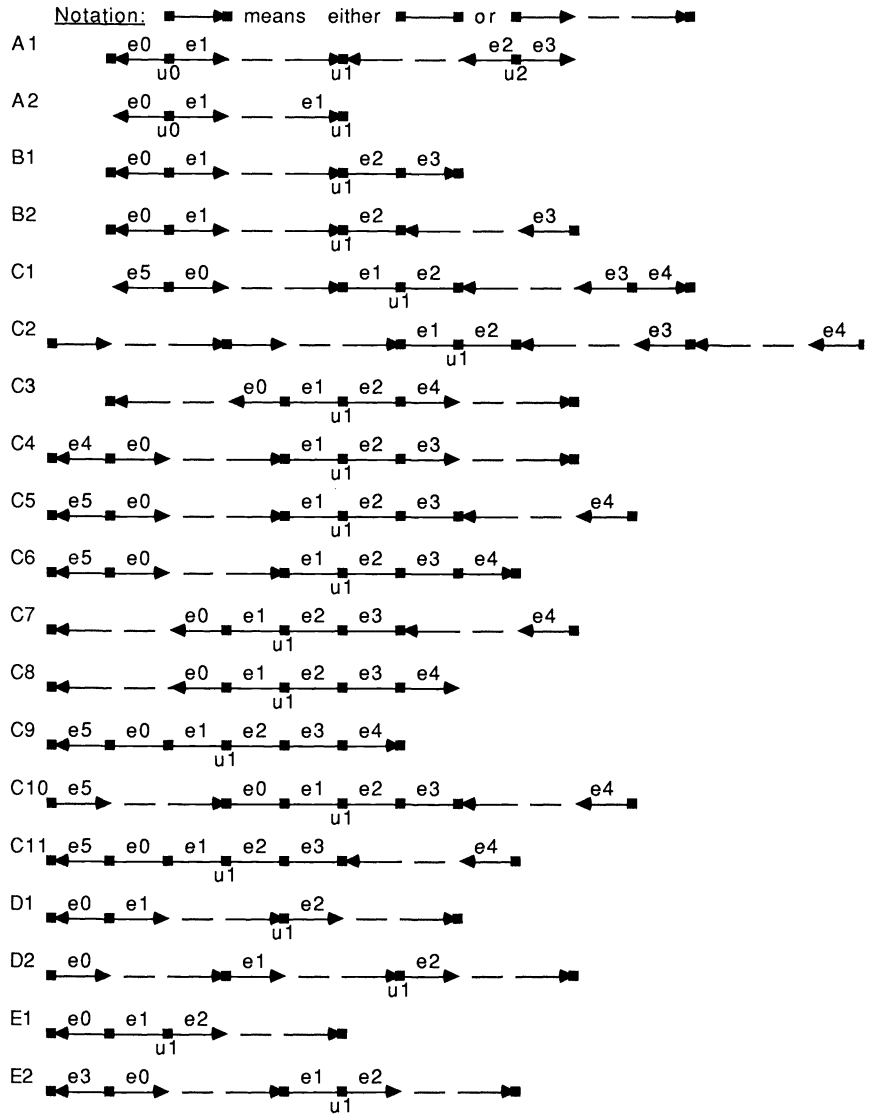


FIG. 11. *Classification of files sent at* $t = 0$.

Proof. We consider all possible situations as classified in Fig. 11. Let f_i, $1 \leqq i \leqq 5$ be finishing times and let $e_j e_k$ mean e_j and e_k are transmitted at $t = 0$.

For nodes of type A:

$$A_1(f_1, f_2, f_3, f_4) \text{ correspond to } (e_1e_2, e_1e_3, e_0e_2, e_0e_3),$$

$$A_2(f_1, f_2) \text{ correspond to } (e_0, e_1).$$

Clearly these consider all possible delays for e_1 and e_2.

For nodes of type B:

$$B_1(f_1, f_2, f_3, f_4) \text{ correspond to } (e_1e_2, e_1e_3, e_0e_2, e_0e_3),$$

$$B_2(f_1, f_2) \text{ correspond to } (e_1e_2, e_0e_2).$$

For nodes of type C:

$C_1(f_1, f_2, f_3, f_4)$ correspond to $(e_1e_3, e_1e_4, e_0e_2, e_5e_2)$,

$C_2(f_1, f_2)$ correspond to (e_1e_3, e_0e_2),

$C_3(f_1, f_2, f_3)$ correspond to (e_1e_4, e_0e_2, e_0e_4),

$C_4(f_1, f_2, f_3)$ correspond to (e_1e_3, e_0e_2, e_4e_2),

$C_5(f_1, f_2, f_3)$ correspond to (e_1e_3, e_0e_2, e_5e_2),

$C_6(f_1, f_2, f_3, f_4)$ correspond to $(e_1e_3, e_1e_4, e_0e_2, e_5e_2)$,

$C_7(f_1, f_2, f_3)$ correspond to (e_1e_3, e_0e_2, e_0e_3),

$C_8(f_1, f_2, f_3, f_4)$ correspond to $(e_1e_3, e_1e_4, e_0e_2, e_0e_3)$,

$C_9(f_1, \cdots, f_5)$ correspond to $(e_1e_3, e_1e_4, e_0e_2, e_5e_2, e_0e_3)$,

$C_{10}(f_1, f_2, f_3)$ correspond to (e_1e_3, e_0e_2, e_0e_3),

$C_{11}(f_1, f_2, f_3, f_4)$ correspond to $(e_1e_3, e_0e_2, e_0e_3, e_5e_2)$.

For nodes of type D always schedule e_2 first by Lemma 4:

$D_1(f_1, f_2)$ correspond to (e_1e_2, e_0e_2),

$D_2(f_1)$ correspond to (e_1e_2).

For nodes of type E:

$E_1(f_1, f_2)$ correspond to (e_1, e_0e_2),

$E_2(f_1, f_2, f_3)$ correspond to (e_1, e_0e_2, e_3e_2).

In each case it is not hard to show that once the chosen files are scheduled at $t=0$, the finishing time at u_1 is determined by scheduling the first ready file at u_1 first or by scheduling arbitrarily (see Lemma 5). Since no vertex has more than five possible finishing times, and each finishing time is determined by only at most two particular files starting at $t=0$ the result follows.

We therefore obtain the following algorithm which will decide, given deadline d, whether there is a feasible schedule for G meeting the deadline, where G is a path or a cycle. For each e_j we associate a literal x_j. Then x_j is true if and only if e_j is scheduled at $t=0$.

ALGORITHM D.

(1) For each node of type A, B, C, D, E calculate the possible finishing times, as outlined in Lemma 6. If $d < f_i$ for any f_i and f_i corresponds to e_je_k then generate $\neg(x_jx_k) \equiv (\bar{x}_j + \bar{x}_k)$. If f_i corresponds to e_j then generate $\bar{x}_j$.

(2) For each node u_1 of types (i), (ii), (iii) of Fig. 9(a) generate $(\bar{x}_0 + \bar{x}_1)$. Use the algorithm given in [5] to see if the generated collection of clauses C is satisfiable. Note that each clause has at most two literals, and that there are $O(|V|)$ literals.

(3) If C is satisfiable then C has a feasible schedule meeting deadline d. For each x_i such that x_i is assigned to be true, schedule e_i at $t=0$. If C is not satisfiable no schedule exists.

(4) For each remaining node not scheduled above, schedule files as soon as they are ready. Ties may be scheduled arbitrarily.

THEOREM 7. *If G is a path or cycle in which lengths of files are arbitrary, then G is scheduled with deadline d by Algorithm* D, *in time $O(|E|)$. If Algorithm* D *fails then no schedule is possible.*

Proof. Step (1) of Algorithm D checks the finishing times that are possible at each branch node that is a terminal node for some file, or is adjacent to another branch node. If each of these nodes finishes within the given deadline, then it is clear that the schedule is feasible, otherwise it is not feasible. In the reduction to 2-SAT we must show that FTPF is solvable if and only if the instance of 2-SAT is solvable.

Let C be the instance of 2-SAT. Suppose a feasible schedule exists. Each of the files e_j started at $t=0$ corresponds to x_j is true. Similarly, if e_j is not started at $t=0$ then x_j is false. Since the deadline d is met at each node, this means $d \geqq f_i$ for all e_j, e_k started at $t=0$. Hence for each finishing time f_i such that $d < f_i$ the files corresponding to f_i are not both started at $t=0$ or, in the case of only one file, that file is not started at $t=0$. In the case of two files, e_j, e_k, $(\bar{x}_j + \bar{x}_k)$ is true. In the case of one file e_j we have that $\bar{x}_j$ is true. Hence each clause in step (1) is satisfied. Since we have a valid schedule, each clause in step (2) holds. Hence C is satisfied.

Conversely, suppose that C is satisfied. Then the schedule constructed by the truth assignment given in Algorithm D is feasible. Suppose not, for a contradiction, then some node of type A, B, C, D, E does not meet the deadline d, i.e., $f_i > d$ for some finishing time f_i. (It is clear that two files will not be started at time $t=0$ from the same node by the clauses generated in Step (2)). Hence there exist files e_j and e_k such that e_j and e_k are both started at $t=0$, or a file e_l such that e_l is started at $t=0$. But Algorithm D generates $(\bar{x}_j + \bar{x}_k)$ in C, and $\bar{x}_l$ in C which implies that x_j and x_k are not both true, or x_l is not true. Therefore the schedule generated by Algorithm D does not have both e_j and e_k starting at $t=0$, nor e_l starting at $t=0$. This contradiction shows that the original schedule must be feasible.

Finally, we note that steps (1) and (2) of Algorithm D take $O(|V|)$ time (see [5]), and steps (3) and (4) take time $O(|E|)$. We note that step (2) creates any additional clauses needed for cycles.

COROLLARY 1. *An optimal algorithm may be obtained in time* $O(|E| \cdot \log |E|)$. *An optimal algorithm may be obtained for* $\alpha(v) > 1$, *with the same time bound, as outlined in* [4].

We conclude with an example of how Algorithm D produces a feasible schedule. Consider the path shown in Fig. 12 with the possible finishing times calculated at each node. We consider first $d=6$. The finishing times that do not meet the deadline correspond to the following files:

$$e_1, e_0e_2, e_0e_3, e_3e_5, e_2e_4, e_2e_5, e_4e_7, e_6.$$

Step (1) of Algorithm D generates the clauses

$$\bar{x}_1(\bar{x}_0+\bar{x}_2)(\bar{x}_0+\bar{x}_3)(\bar{x}_3+\bar{x}_5)(\bar{x}_2+\bar{x}_4)(\bar{x}_2+\bar{x}_5)(\bar{x}_4+\bar{x}_6)(\bar{x}_4+\bar{x}_7)\bar{x}_6.$$

Since this is not satisfiable no feasible schedule is possible.

For deadline $d=7$, the collection of clauses C in Algorithm D is satisfiable with truth assignment given by

$$x_0 = \bar{x}_1 = \bar{x}_2 = x_3 = x_4 = \bar{x}_5 = x_6 = \bar{x}_7 = \textit{true}.$$

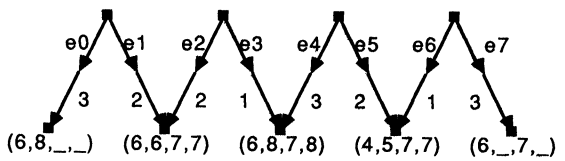


FIG. 12. *Example of the schedule produced by Algorithm* D.

This corresponds to starting the files e_0, e_3, e_4, e_6 at $t=0$. From the calculations of the finishing times shown in Fig. 12 it is clear that this yields a valid schedule.

Appendix. We now prove the technical lemma needed in Proposition 1.

LEMMA A1. *The truth assignment for u_i is consistent.*

Proof. If T is any task and S is a schedule, then let $t(T)$ be the assigned start time of T in S, where $j=p_i$ "+," is defined by $j+1=1$. We show that for each j if $u_{i\theta_i(j)}$ is scheduled according to u_i is true, then so is $u_{i\theta_i(j+1)}$.

(I) Suppose $u_i \in c_{\theta_i(j)}$.

(1) j even. Then $u_{i\theta_i(j)} < a_{i\theta_i(j)}$ and $a_{i\theta_i(j+1)} < u_{i\theta_i(j+1)}$. Now if u_i is true then $t(u_{i\theta_i(j)}) = s_1(u_{i\theta_i(j)})$.

(a) $\bar{u}_i \in c_{\theta_i(j+1)}$. By the scheduling of $d_{i\theta_i(j)}$ we have

$$t(d_{i\theta_i(j)}) = s_2(u_{i\theta_i(j+1)}) \Rightarrow t(u_{i\theta_i(j+1)}) = s_1(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(b) $u_i \in c_{\theta_i(j+1)}$. By the scheduling of $d_{i\theta_i(j)}$ we have

$$t(d_{i\theta_i(j)}) = s_1(a_{i\theta_i(j+1)}) \Rightarrow t(a_{i\theta_i(j+1)}) = s_2(a_{i\theta_i(j+1)})$$
$$\Rightarrow t(u_{i\theta_i(j+1)}) = s_2(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(2) j odd. Then we have $a_{i\theta_i(j)} < u_{i\theta_i(j)}$ and $u_{i\theta_i(j+1)} < a_{i\theta_i(j+1)}$. If $u_i = \textit{true}$ then $t(u_{i\theta_i(j)}) = s_2(u_{i\theta_i(j)})$.

(a) $\bar{u}_i \in c_{\theta_i(j+1)}$. By scheduling of $d_{i\theta_i(j)}$

$$t(d_{i\theta_i(j)}) = s_1(u_{i\theta_i(j+1)}) \Rightarrow t(u_{i\theta_i(j+1)}) = s_2(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(b) $u_i \in c_{\theta_i(j+1)}$. By the scheduling of $d_{i\theta_i(j)}$

$$t(d_{i\theta_i(j)}) = s_2(a_{i\theta_i(j+1)}) \Rightarrow t(a_{i\theta_i(j+1)}) = s_1(a_{i\theta_i(j+1)})$$
$$\Rightarrow t(u_{i\theta_i(j+1)}) = s_1(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(II) Suppose $\bar{u}_i \in c_{\theta_i(j)}$.

(1) j even. Then we have $u_{i\theta_i(j)} < a_{i\theta_i(j)}$ and $a_{i\theta_i(j+1)} < u_{i\theta_i(j+1)}$. If $u_i = \textit{true}$ then $t(u_{i\theta_i(j)}) = s_2(u_{i\theta_i(j)}) \Rightarrow t(a_{i\theta_i(j)}) = s_2(a_{i\theta_i(j)})$ by the precedence constraints.

(a) $\bar{u}_i \in c_{\theta_i(j+1)}$. Then by the scheduling of $d_{i\theta_i(j)}$ we have

$$t(d_{i\theta_i(j)}) = s_2(u_{i\theta_i(j+1)}) \Rightarrow t(u_{i\theta_i(j+1)}) = s_1(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(b) $u_i \in c_{\theta_i(j+1)}$. Then we have

$$t(d_{i\theta_i(j)}) = s_1(a_{i\theta_i(j+1)}) \Rightarrow t(a_{i\theta_i(j+1)}) = s_2(a_{i\theta_i(j+1)})$$
$$\Rightarrow t(u_{i\theta_i(j+1)}) = s_2(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(2) j odd. Then we have $a_{i\theta_i(j)} < u_{i\theta_i(j)}$ and $u_{i\theta_i(j+1)} < a_{i\theta_i(j+1)}$. If $u_i = \textit{true}$ then $t(u_{i\theta_i(j)}) = s_1(u_{i\theta_i(j)}) \Rightarrow t(a_{i\theta_i(j)}) = s_1(a_{i\theta_i(j)})$.

(a) $\bar{u}_i \in c_{\theta_i(j+1)}$. Then

$$t(d_{i\theta_i(j)}) = s_1(u_{i\theta_i(j+1)}) \Rightarrow t(u_{i\theta_i(j+1)}) = s_2(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

(b) $u_i \in c_{\theta_i(j+1)}$. Then

$$t(d_{i\theta_i(j)}) = s_2(a_{i\theta_i(j+1)}) \Rightarrow t(a_{i\theta_i(j+1)}) = s_1(a_{i\theta_i(j+1)})$$
$$\Rightarrow t(u_{i\theta_i(j+1)}) = s_1(u_{i\theta_i(j+1)}) \leftrightarrow u_i = \textit{true}.$$

Since j is arbitrary, $1 \leqq j \leqq p_i$, we conclude that the truth assignment for u_i is consistent, and the lemma is proved.

We conclude with the proof of the lemma needed for Theorem 2.

LEMMA A2. *Given a node v, deadline d, and a set of available times*

$$S = \{s_1, s_2, \cdots, s_k\} \subseteq \{0, 1, \cdots, d-1\},$$

there exists an FTPF *problem represented as a tree G with v as the root, with all lengths equal to one, such that if the subtree below v can be scheduled with deadline d, the remaining set of available times for v is S.*

Proof. For each $j \in \{0, 1, \cdots, d-1\}$ such that $j \notin S$ create a child node v_j of v such that the edge from v_j to v represents a file transmission of length 1. We construct a subtree of v_j so that this file may only be sent to v at time j for a feasible schedule of deadline d.

Suppose d is odd. Then if j is odd we have that $0, 1, \cdots, j-1$ is an odd number of time slots and so is $j+1, \cdots, d-1$. Create a directed path $v_j, v_{0,1}, \cdots, v_{0,d}$ representing a file that must be forwarded d times starting at v_j. For a feasible schedule with deadline d, this file must be sent at time $t=0$ from v_j. For each pair $(k, k+1)$ where k is odd and $k-1 \leqq j-1$ create a directed path $v_{k,0}, \cdots, v_{k,k}, v_j, v_{k,k+2}, \cdots, v_{k,d}$. This represents a file that must be forwarded d times, and to attain a schedule with deadline d must be scheduled at v at times k and $k+1$. Similarly, for time slot $d-1$ create a directed path $v_{d-1,0}, \cdots, v_{d-1,d-1}, v_j$ that must be scheduled at time $d-1$ at v_j. For the pairs $(k, k+1)$ where k is even, $k \leqq j+1 \leqq d-3$, we may create directed paths as described above in which a file must be scheduled at v_j during times k and $k+1$.

If d is odd and j is even, then the time slots $0, 1, \cdots, j-1$ and $j+1, \cdots, d-1$ are each an even number. We create directed paths that represent files that must be scheduled at v_j at times k and $k+1$, where $0 \leqq k \leqq j-2$, k even, and $j+1 \leqq k \leqq d-2$, k odd.

The case for d even is similar and we omit the proof.

We conclude that if there is a feasible schedule at v with deadline d, then the set of available times for which v is not busy is exactly S.

REFERENCES

[1] H. A. CHOI AND S. L. HAKIMI, *Scheduling data transfers in networks with preemptions*, in Proc. 23 Allerton Conference on Communication, Control, and Computing, Univ. of Illinois, Urbana-Champaign, IL, October 1985.

[2] ———, *Scheduling file transfers for trees and odd cycles*, SIAM J. Comput., 16 (1987), pp. 162-168.

[3] ———, *Data transfers in networks*, to appear.

[4] E. G. COFFMAN, JR., M. R. GAREY, D. S. JOHNSON, AND A. S. LAPAUGH, *Scheduling file transfers*, SIAM J. Comput., 14 (1985), pp. 744-780.

[5] S. EVEN, A. ITAI, AND A. SHAMIR, *On the complexity of timetable and multicommodity flow problems*, SIAM J. Comput., 5 (1976), pp. 691-703.

[6] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractibility: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[7] M. R. GAREY, D. S. JOHNSON, B. B. SIMONS, AND R. E. TARJAN, *Scheduling unit-time tasks with arbitrary release times and deadlines*, SIAM J. Comput., 10 (1981), pp. 256-269.

[8] I. S. GOPAL, G. BONGIOVANNI, M. A. BONUCELLI, D. T. TANG, AND C. K. WONG, *An optimal switching algorithm for multibeam satellite systems with variable bandwidth beams*, IEEE Trans. Comm., 30, 11 (1982), pp. 2475-2481.

[9] B. HAJEK, *Link schedules, flows and the multichromatic index of graphs*, in Proc. 1984 Conference on Information Sciences and Systems, Princeton, NJ, 1984, pp. 498-502.

[10] B. HAJEK AND G. SASAKI, *Link schedules in polynomial time*, Department of Electrical and Computer Engineering and the Coordinated Science Laboratory, University of Illinois, Urbana, IL, preprint, 1985.

[11] T. LANG AND E. B. FERNANDEZ, *Scheduling of unit-length independent tasks with execution constraints*, Inform. Process. Lett., 4 (1976), pp. 95-98.

[12] K. NAKAJIMA AND S. L. HAKIMI, *Complexity results for scheduling tasks with discrete starting times*, J. Algorithms, 3 (1982), pp. 344-361.

[13] ———, *On scheduling equal execution time tasks with discrete starting times*, in Proc. 16 Annual Conference on Information Sciences and Systems, 1982, pp. 75-79.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 246-255, April 1990

003

CHARACTERIZATION OF ASSOCIATIVE OPERATIONS WITH PREFIX CIRCUITS OF CONSTANT DEPTH AND LINEAR SIZE*

G. BILARDI† AND F. P. PREPARATA‡

Abstract. The *prefix problem* consits of computing all the products $x_0x_1 \cdots x_j$ $(j=0, \cdots, N-1)$, given a sequence $\mathbf{x}=(x_0, x_1, \cdots, x_{N-1})$ of elements in a semigroup. It is shown that there are unbounded fan-in and fan-out *Boolean circuits* for the prefix problem with *constant depth* and *linear size* if and only if the Cayley graph of the semigroup does not contain a special type of cycle called *monoidal cycle.*

Key words. prefix computation, Boolean circuits, size-depth trade-offs, semigroups, superconcentrators

AMS(MOS) subject classifications. 68Q10, 68Q20, 20M99, 94C99

1. Introduction.

The *prefix problem* consists of computing all the products $x_0x_1 \cdots x_j$ $(j=0, \cdots, N-1)$, given a sequence $\mathbf{x}=(x_0, x_1, \cdots, x_{N-1})$ of elements in a semigroup. Prefix computations occur in the solution of several significant problems such as the evolution of finite-state machines [O63], [LF80], binary addition [O63], [BK82], linear recurrences [K78], digital filtering [BP86], various graph problems [KRS85], sorting in bit-models of computation [CS85], [BP85], and scheduling [DS83]. Several built or proposed parallel computers include some variant of the prefix operation as a basic instruction, e.g., fetch-and-add on the Ultracomputer [GGKMRS83], scan on the Connection Machine [Hl85], and multiprefix on the Fluent Machine [Rn89].

The prefix problem has been extensively investigated (see [P87] for a survey). Various complexity measures such as *size, depth, width,* and their trade-offs have been studied in [LF80], [F83], and [S86] for *circiuts of semigroup multipliers.* Algorithms for the EREW-PRAM model have been proposed in [KRS85], [R84], and [CV86]. Implementations on a *tree-connected* network are discussed in [DS83]. *Size-time* trade-offs for *Boolean networks,* which are synchronized interconnections of Boolean gates and one-bit storage devices, have been characterized in [BP87]. *Size-depth* trade-offs for *Boolean circuits* with NOT, OR, and AND gates of unbounded fan-in and fan-out have been studied in [CFL83a] and [CFL83b] and will also be the subject of this paper.

It is interesting to observe that the prefix problem reveals different facets in different models of computation. The structure and the complexity of prefix circuits are oblivious to the nature of the underlying semigroup if semigroup multipliers are assumed as the primitive modules, whereas they depend on the nature of the semigroup if the primitive modules are Boolean gates.

In the present paper we study the size of prefix Boolean circuits of constant depth, a subject whose investigation began in [CFL83a] and [CFL83b]. The goal is the identification of the semigroup properties that determine the size complexity.

In [CFL83a] it was shown that the size of constant-depth prefix circuits for nongroup-free semigroups is exponential, as a simple consequence of known bounds on the size of constant-depth circuits for the parity function [FSS81], [Y85], [H86].

* Received by the editors April 13, 1988; accepted for publication (in revised form) June 5, 1989.

† Department of Computer Science, Cornell University, Ithaca, New York 14853. This research was partially supported by the National Science Foundation under grant MIP-86-02256 and by the Joint Services Electronics Program under contract F49620-8-C0044.

‡ Coordinated Science Laboratory, Department of Electrical and Computer Engineering and Department of Computer Science, University of Illinois, Urbana-Champaign, Illinois 61801. This research was partially supported by the National Science Foundation under grant ECS-84-10902.

For group-free semigroups, a construction is provided that achieves slightly superlinear size. Indeed, given any primitive recursive function $f(N)$, the construction can be tuned to achieve size $O(Nf^{-1}(N))$ and depth independent of N (but dependent upon f and upon the underlying semigroup).

In [CFL83b] the above bound is shown to be tight for the *carry semigroup*, whose prefix circuit computes the carries in binary addition. The lower bound is obtained by showing that any prefix circuit for the carry semigroup contains a *weak superconcentrator*, and hence satisfies a lower bound on the size of constant-depth weak superconcentrators due to [DDPW83]. Finally, [CFL83b] show that not all group-free semigroups require superlinear size by constructing a linear-size prefix circuit for the OR semigroup.

Given these results, it is natural to ask which semigroups have linear-size constant-depth prefix circuits. In this work, we introduce the notion of *monoidal cycle*, a special type of cycle in the Cayley graph of a semigroup, and we show that there are constant-depth linear-size prefix circuits for a semigroup if and only if its Cayley graph has no monoidal cycle.

The remainder of this paper is organized as follows. Section 2 contains the necessary technical definitions about semigroups. In particular, two distinct elements a and b of a semigroup are said to form a monoidal cycle if and only if, for some c, d, and ω in the semigroup, $ac = b$, $bd = a$, $a\omega = a$, and $b\omega = b$. Section 3 contains a superlinear lower bound for semigroups with monoidal cycles. This result is obtained by showing that any prefix circuit for a semigroup with monoidal cycles contains an α-weak superconcentrator graph, a generalization of a weak superconcentrator with similar size-depth trade-offs. It should be observed that the lower-bound argument of [CFL83b] cannot be applied directly to arbitrary semigroups with monoidal cycles because it crucially exploits the special nature of the monoidal cycle of the carry semigroup, where $c = b$ and $d = a$. Our lower bound matches the upper bounds of [CFL83a], for the class of group-free semigroups with monoidal cycles.

Section 4 investigates semigroups without monoidal cycles. It is first shown that the absence of monoidal cycles implies some more global algebraic properties. These properties are then exploited in a nontrivial construction that yields prefix circuits of constant depth and linear size. Finally, in § 5, we compare the classification of semigroups with reference to the size complexity of constant-depth prefix circuits with the classification of semigroups with reference to the size-time complexity of prefix Boolean networks [BP89].

As pointed out by Pippenger [P], a construct equivalent to the monoidal cycle—as introduced in this paper—had been considered earlier in automata theory. Indeed, the absence of such construct is characteristic of the syntactic monoid of the so-called "irreversible nets." (See Problem 19 in Chap. 5 and Problem 17 in Chap. 6 of [MP71], and [Z70].)

2. Preliminaries. A *finite semigroup* is a pair $\langle A, \cdot \rangle$ where $A = \{a_1, a_2, \cdots, a_s\}$ is a set of *size* s and $\cdot$ is an *associative* binary operation on A, which we call *product*. We denote by xy the product of elements $x, y \in A$. A *finite monoid* is a finite semigroup with a distinguished element e, called the *identity*, such that $xe = ex = x$, for all $x \in A$.

For a sequence $\mathbf{x} = (x_0, x_1, \cdots, x_{N-1}) \in A^N$, the sequence of *prefixes* of $\mathbf{x}$ is defined as $\mathbf{y} = (y_0, y_1, \cdots, y_{N-1})$, with $y_j = x_0 x_1 \cdots x_j$. The *prefix problem* consists in computing $\mathbf{y}$ from $\mathbf{x}$.

In the study of the prefix problem, an important role is played by the *Cayley graph* $G(A) = (A, E)$ of A, containing for each ordered pair (x, y) an arc of the form (x, xy), labeled by y.

An element a of A is said to be *recurrent* if $G(A)$ contains a self-loop at a, that is, if there is an element b of a, not necessarily distinct from a, such that $ab = a$.

Two elements $a, b \in A$ are said to be *equivalent*, denoted $a \equiv b$, if there are $c, d \in A$ such that $ac = b$ and $bd = a$, i.e., a and b are joined in a cycle in $G(A)$. Note "$\equiv$" is an equivalence relation; its equivalence classes are the strongly connected of components of $G(A)$.

A semigroup $\langle A, \cdot \rangle$ has a *monoidal cycle* if there are five elements, a, b, c, d, and ω in A (not all necessarily distinct but with $a \neq b$) such that $b = ac$, $a = bd$, $a\omega = a$, and $b\omega = b$. In other words, the Cayley graph $G(A)$ contains a two-node cycle with nodes a and b having identically labeled self-loops. A semigroup is referred to as MC or NMC depending upon whether or not it has monoidal cycles, respectively. We will see that monoidal cycles are crucial in determining the complexity of the prefix problem.

It is well known (and also easy to prove) that for each element a in a finite semigroup there are two positive integers k and p such that $a, a^2, \cdots, a^{k+p-1}$ are all distinct, and $a^{k+p} = a^k$. Moreover, if the *period* p of a is larger than 1, then $\{a^k, a^{k+1}, \cdots, a^{k+p-1}\}$ form a group. If $p = 1$ for all the elements, then the semigroup is called *group-free* [MP71].

We now give examples of semigroups that belong to the various classes introduced above. If any element $x \in A$ different from the identity has an inverse x^{-1} such that $xx^{-1} = e$, then (e, x, e) forms a monoidal cycle in $G(A)$ and A is MC. As a corollary, all *groups* are MC.

A special but important case of NMC semigroups are the *cycle-free* (CF) semigroups, defined by the absence of cyles in the Cayley graph. We denote a noncycle-free semigroup as NCF. An example of CF semigroup is the *left-zero* semigroup $\langle L_p, \circ \rangle$, where $L_p = \{l_1, l_2, \cdots, l_p\}$ and $l_i \circ l_j = l_i$, for all l_i and l_j. An example of semigroup that is NMC and NCF is the *right-zero* semigroup $\langle R_q, * \rangle$, where $R_q = \{r_1, r_2, \cdots, r_q\}$ and $q \geqq 2$, and $r_i^* r_j = r_j$, for all r_i and r_j. This semigroup with $q = 2$ and with the adjunction of the identity, becomes the monoid that models the function "carry" in binary addition (the identity representing carry propagation and the two zeros representing carry setting and carry resetting, respectively). With the identity, the semigroup becomes MC, although it remains group-free.

Further examples of CF semigroups are all *semilattices*, where the semigroup operation is commutative and idempotent. Semilattices include the set of the 0-1 vectors of length n with respect to component-wise OR (AND), and the set of the first s nonnegative integers with the MINIMUM (MAXIMUM) operation.

3. Lower bound. As the model of computation, we consider *circuits* defined in terms of the underlying graphs, as follows. Let G be a directed acyclic graph. Inputs are the vertices of indegree 0, and outputs are the vertices of outdegree 0. Each vertex distinct from an input is labeled by an arbitrary Boolean function with an aribtrary number of inputs and one output, and represents a *gate* computing that function. All gates distinct from the outputs have unbounded outdegree (fan-out). The *size* of the circuit is the number of arcs of its underlying graph; its *depth* is the number of arcs in the longest path from input to output. We observe that the lower bound derived in this section is independent of the nature of the function computed by the vertices of the circuit. The explicit construction given in the next section, however, assumes the standard base {AND, OR, NOT}.

We now define the notion of α-weak superconcentrator. Let i_h, j_h, $h = 1, \cdots, k$, be integers in the range $\{1, 2, \cdots, N\}$. An *interleaved matching* of size k on $\{1, 2, \cdots, N\}$ is a sequence of pairs $((i_1, j_1)(i_2, j_2), \cdots, (i_k, j_k))$, such that $i_1 < j_1 < i_2 <$

$j_2 < \cdots < i_k < j_k$. Let $\hat{G}$ be a directed acyclic graph with N vertices $\{u_1, \cdots, u_N\}$ of indegree 0 and N vertices $\{v_1, \cdots, v_N\}$ of outdegree 0. For a fixed constant $0 < \alpha \leqq 1$, we say that $\hat{G}$ is an *α-weak N-superconcentrator* if, for any interleaved matching $((i_1, j_1), \cdots, (i_k, j_k))$ of size k, $\hat{G}$ contains at least αk vertex-disjoint paths joining $\{u_{i_1}, \cdots, u_{i_k}\}$ and $\{v_{j_1}, \cdots, v_{j_k}\}$. A 1-weak superconcentrator is a weak superconcentrator as defined in [DDPW83].

The size-depth trade-off of weak superconcentrators can be expressed in terms of the functions f_i, $i \geqq 1$, defined as $f_1(n) = 2^n$, and $f_{i+1}(n) = f_i^{(n)}(2)$, where $f^{(n)}$ denotes the n-fold iterate of f. Each f_i is primitive recursive, monotone, and grows faster than f_{i-1}. Moreover, each primitive recursive function is asymptotically bounded above by a suitable f_i. If f is monotone increasing, $f^{-1}(n)$ denotes the least k such that $f(k) \geqq n$.

Our interest in α-weak N-superconcentrators is due to the following result, whose proof is a minor adaptation of the proof of Theorem 2 in [DDPW83] and is therefore omitted here.

LEMMA 1. *The size of an α-weak N-superconcentrator of depth exactly $2d$ is $\Omega(Nf_d^{-1}(N))$.*

We shall also need the following lemma, essentially due to [V76], which relates a graph-theoretic property of a circuit to a property of the function computed by it.

LEMMA 2. Let A be a set of input vertices, *and let B be a set of output vertices of a circuit C. If, for some fixed values of the variables not applied to A, there are 2^q assignments to the (binary) variables applied to A each causing a distinct configuration for the variables available at B, then there are at least q vertex-disjoint paths from vertices of A to vertices of B.*

Proof. Each set of vertices of C whose removal disconnects B from A must have cardinality at least q since the values computed by these vertices uniquely determine the values computed by the vertices in B. The claim then follows from Menger's theorem. □

We are now ready to state the main result of this section.

THEOREM 1. *If $\langle A, \cdot \rangle$ is an* MC *semigroup, then any length-N prefix circuit for $\langle A, \cdot \rangle$ of depth d has size $\Omega(Nf_{\lfloor (d+1)/2 \rfloor}^{-1}(N))$.*

Proof. We shall construct a $(\frac{1}{2})$-weak $(N-1)$-superconcentrator C by modifications of the prefix circuit for $\langle A, \cdot \rangle$, which increase the depth by one and the size by $O(N)$. Then, the result follows from Lemma 1.

Throughout this proof, a and b form a monoidal cycle such that $a\omega = a$, $b\omega = b$, $ac = b$, and $bd = a$.

In the prefix circuit, x_i and y_i, respectively, denote the variables applied to the ith input and output terminals ($i = 0, 1, \cdots, N-1$). Values of input and output semigroup variables are encoded in binary. Specifically, we let x_{ij}, $j = 1, 2, \cdots, r(i)$, be the binary variables encoding $x_i \in A$ and y_{ij}, $j = 1, 2, \cdots, s(i)$, the binary variables encoding $y_i \in A$. Note that we allow different encodings to be used for different input/output variables. We let $x_{ij}(z)$ be the value of x_{ij} in the encoding of semigroup element z. Likewise we define $y_{ij}(z)$.

We now construct a circuit C by adding to the prefix circuit $(N-1)$ new binary terminals $u_1, \cdots, u_{N-1}$ and $p_1, \cdots, p_{N-1}$, connected as follows. For each $i = 1, 2, \cdots, N-1$ and each $j = 1, 2, \cdots, r(i)$, the input vertex for variable x_{ij} is replaced by a vertex with inputs u_i and p_i, which computes the function

$$x_{ij} = p_i(\neg u_i x_{ij}(\omega) \vee u_i x_{ij}(c)) \vee \neg p_i(\neg u_i x_{ij}(\omega) \vee u_i x_{ij}(d)).$$

Intuitively, if $p_i = 1$, then $u_i = 0$ forces $x_i = \omega$ and $u_i = 1$ forces $x_i = c$. If $p_i = 0$, then $u_i = 0$ forces $x_i = \omega$ and $u_i = 1$ forces $x_i = d$.

Since $a \neq b$, there exists a smallest j such that $y_{ij}(a) \neq y_{ij}(b)$, and we choose v_i as the terminal that outputs y_{ij} in C, for $i = 1, \cdots, N-1$. We shall prove that C is a $(\frac{1}{2})$-weak $(N-1)$-superconcentrator with respect to the sets of vertices $U = \{u_1, \cdots, u_{N-1}\}$ and $V = \{v_1, \cdots, v_{N-1}\}$.

For simplicity, let k be an even number, and let $((i_1, j_1), \cdots, (i_k, j_k))$ be an interleaved matching. We consider the set of assignments to the inputs of C satisfying the following conditions.

(1) $x_0 = a$.

(2) For $i \notin \{i_1, \cdots, i_k\}$, $u_i = p_i = 0$, so that $x_i = \omega$.

(3) For odd h, $p_{i_h} = 1$, $p_{i_{h+1}} = 0$, and $u_{i_h} = u_{i_{h+1}}$.

Clearly, there are $2^{k/2}$ such assignments. It is also easy to see that in the output configurations corresponding to such assignments, for odd h we have $v_{i_h} = u_{i_h}$. From Lemma 2, with $q = k/2$, we conclude that there are at least $k/2$ vertex-disjoint paths from inputs to outputs of the chosen interleaved matching. In conclusion, C is an $(1/2)$-weak superconcentrator as claimed. □

Theorem 1 shows that the prefix circuits proposed in [CFL83a] for group-free semigroups, which have size $O(Nf_d^{-1}(N))$ and depth $O(d)$, have optimal size to within a constant factor for the depth, when the group-free semigroup is in particular an MC semigroup.

4. Upper bound. In this section we give a constructive proof that if $\langle A, \cdot \rangle$ is an NMC-semigroup, then there is a constant-depth prefix circuit for $\langle A, \cdot \rangle$ of linear size. We begin by deriving a crucial property of $G(A)$ as a direct consequence of the absence of monoidal cycles.

LEMMA 3. *Let x be a recurrent element of A, and, for some u and v in A, let $ux = a$ and $vx = b$, with $a \equiv b$. If $\langle A, \cdot \rangle$ is an* NMC*-semigroup, then $a = b$.*

Proof. Assume, for a contradiction, that $a \neq b$. For any $c \in A$, let $SL(c) \triangleq \{\omega \mid c\omega = c\}$, i.e., the set of labels of the self-loops of c in $G(A)$. Since x is recurrent, then $|SL(x)| \geq 1$. We claim that $SL(x) \subseteq SL(a)$. Indeed, for any $\omega \in SL(x)$, $x\omega = x$; using the hypothesis $ux = a$, we have $a\omega = (ux)\omega = u(x\omega) = ux = a$, i.e., $a\omega = a$, as claimed. Analogously, we show $SL(x) \subseteq SL(b)$. But the absence of monoidal cycles means that for two equivalent elements a and b, we have $SL(a) \cap SL(b) = \emptyset$. This implies $SL(x) = \emptyset$, contrary to the hypothesis that x is recurrent. □

Let m' be the number of nodes on the longest path of $G(A)$ consisting exclusively of nonrecurrent elements of A. (Note that $m' = 0$ if every $a \in A$ is recurrent.) We now establish the following important property of NMC-semigroups:

THEOREM 2. *Let $\langle A, \cdot \rangle$ be an* NMC*-semigroup, and let a, b, c, and d be equivalent elements of A. Let $av_1v_2 \cdots v_{m'+1} = c$ and $bv_1v_2 \cdots v_{m'+1} = d$, where $v_i \in A$, $1 \leq i \leq m'+1$. Then $c = d$.*

Proof. Since $c \equiv a$ by hypothesis, then $av_1v_2 \cdots v_j \equiv a$ for all $j \in [1, m'+1]$, by the definition of strongly connected component of $G(A)$; similarly, we establish $bv_1v_2 \cdots v_j \equiv b$. Moreover, according to the above definition of m', the sequence of the prefixes of the sequence $(v_1, v_2, \cdots, v_{m'+1})$ contains at least one recurrent element. Let $w_k = v_1v_2 \cdots v_k$ $(k \leq m'+1)$ be one such prefix. Since w_k is recurrent and $aw_k \equiv a \equiv b \equiv bw_k$, by Lemma 3 we have $aw_k = bw_k$. Then it trivially follows that $c = av_1 \cdots v_{m'+1} = aw_kv_{k+1} \cdots v_{m'+1} = bw_kv_{k+1} \cdots v_{m'+1} = bv_1 \cdots v_{m'+1} = d$. □

This theorem shows that the mere knowledge that ax and a are in the same strongly connected component of $G(A)$ for some recurrent x dispenses us from knowing exactly the value of a if we wish to know ax.

Before proceeding to the construction of the prefix circuit, we introduce an

important structural parameter of $G(A)$, denoted h_A, and defined as the maximum number of distinct strongly connected components visited by any (directed) path in $G(A)$.

The prefix circuit is best described as an $h_A \times N$ array. The rows of this array are referred to as "stages" numbered from 1 to h_A. Each stage contains N modules, numbered from 0 to $N-1$, responsible for "local" processing, and circuits spanning the entire stage, responsible for global processing. Module $M_j^{(h)}$ appears in the $(j+1)$st column of the hth stage.

Any given instance of prefix computation corresponds to the segmentation of the interval $[0, N-1]$ into a collection of contiguous intervals $\{[j_{i-1}, j_i - 1] \mid i = 1, \cdots, h'\}$, where $0 = j_0 < j_1 < \cdots < j_{h'} = N$, such that $y_{j_{i-1}} \equiv \cdots \equiv y_{j_i - 1}$ for $i = 1, \cdots, h'$ and $y_{j_i - 1} \not\equiv y_{j_i}$, for $i = 1, \cdots, h'-1$. In other words, all prefixes with indices in a given interval are in the same strongly connected component and $h' \leqq h_A$. Referring to Fig. 1, the indices of the N modules in the generic stage h are partitioned into three contiguous intervals: the interval $[0, j_{h-1}]$ of the *passive* modules, the interval $[j_{h-1}+1, j_h]$ of the *productive* modules, and the interval $[j_h + 1, N-1]$ of the *irrelevant modules*. The reasons for these denotations are: A passive module $M_j^{(h)}$ simply passes to $M_j^{(h+1)}$ the (previously correctly computed) prefix y_j it receives from $M_j^{(h-1)}$; a productive module $M_j^{(h)}$ correctly computes prefix y_j and forwards it to $M_j^{(h+1)}$; an irrelevant module $M_j^{(h)}$ computes a possibly incorrect prefix, not to be forwarded to $M_j^{(h+1)}$.

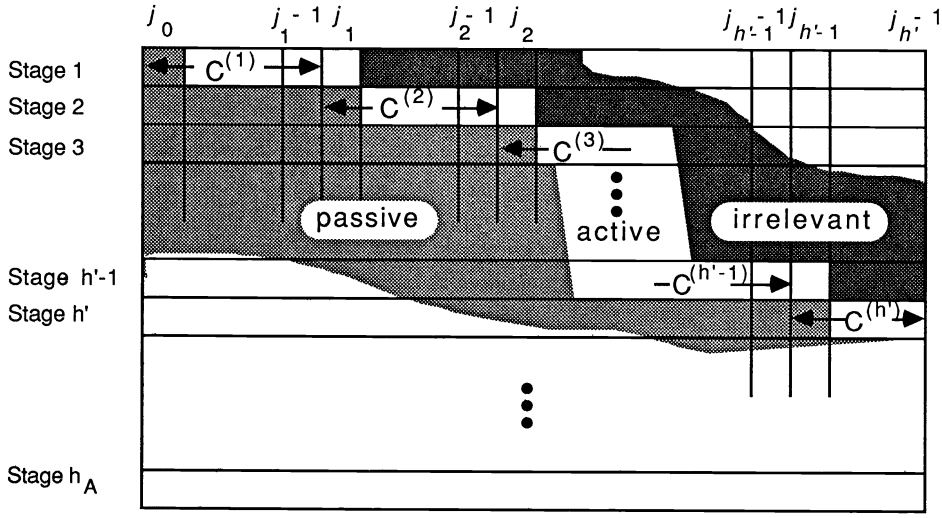


FIG. 1. *Schematic layout of the linear-size prefix circuits for* NMC *semigroups. Each stage is segmented into three portions: the passive modules (left), the productive modules (center), and the irrelevant modules (right).*

We now describe in detail the operation of the generic stage h, where $1 \leqq h \leqq h_A$. Notationally, $u = (u_0, u_1, \cdots, u_{N-1})$ is an N-component vector, where u_j, $0 \leqq j \leqq N-1$, is a member of a finite set B (either $B = A$, the semigroup set, or $B = \{0, 1\}$). Stage h receives three vectors from Stage $h-1$:

(i) The *enable* vector $e^{(h-1)} = (e_0^{(h-1)}, \cdots, e_{N-1}^{(h-1)})$, where $e_j^{(h-1)}$, $0 \leqq j \leqq N-1$, is Boolean. This vector is used to separate the passive modules from the productive and the irrelevant ones, i.e., $e_0^{(h-1)} = e_1^{(h-1)} = \cdots = e_{Jh-1}^{(h-1)} = 0$ and $e_{Jh-1}^{(h-1)} + 1 = \cdots = e_{N-1}^{(h+1)} = 1$.

(ii) The *prefix* vector $y^{(h-1)} = (y_0^{(h-1)}, \cdots, y_{N-1}^{(h-1)})$, where $y_j^{(h-1)}$, $0 \leqq j \leqq N-1$, is an element of A. Of these components, $(y_0^{(h-1)}, \cdots, y_{jh-1}^{(h-1)})$ represent correct values,

i.e., $y_j^{(h-1)} = y_j$ for $j = 0, 1, \cdots, j_{h-1}$ – while $y_j^{(h-1)} = y_{j_{h-1}}$ for $j_{h-1} < j \leqq N-1$. (Note that $y_{j_{h-1}} \neq y_{j_{h-1}-1}$.)

(iii) The *infix* vector $w = (w_0, \cdots, w_{N-1})$, where $w_j \in A$, $0 \leqq j \leqq N-1$. Letting $m \triangleq m' + 2$, the components of w are defined as follows:

$$w_j = \begin{cases} x_0 x_1 \cdots x_j & \text{for } j < m, \\ x_{j-m+1} \cdots x_j & \text{for } j \geqq m. \end{cases}$$

Note that w does not depend on the index h, and is computed by a linear-size constant-depth network. Vectors $e^{(0)}$ and $y^{(0)}$ are correctly initialized as

$$(1) \qquad e^{(0)} = (0, 1, 1, \cdots, 1) \quad \text{and} \quad y^{(0)} = (x_0, x_0, \cdots, x_0).$$

The following algorithm illustrates the operation of Stage h:

(1) $z_j^{(h)} = \begin{cases} w_j & \text{for } j = 0, 1, \cdots, m-1, \\ y_{j-m}^{(h-1)} w_j & \text{for } j = m, m+1, \cdots, N-1. \end{cases}$

(2) $u_{j+1}^{(h)} = \begin{cases} 0 & \text{if } z_j^{(h)} \equiv y_j^{(h-1)} \text{ or } e_j^{(j-1)} = 0, \\ 1 & \text{if } z_j^{(h)} \not\equiv y_j^{(h-1)} \text{ and } e_j^{(h-1)} \end{cases} \quad j = 0, 1, \cdots, N-2.$

(3) $e^{(h)} = \text{PREFIX} - \text{OR}\ (u^{(h)})$.

(4) $t_j^{(h)} = \neg e_j^{(h)} \wedge e_{j+1}^{(h)}$.

(5) $Y^{(h)} = z_{j^*}^{(h)}$ where $t_{j^*}^{(h)} = 1$.

(6) $y_j^{(h)} = \begin{cases} y_j^{(h-1)} & \text{if } e_j^{(h)} = 0, \\ Y^{(h)} & \text{if } e_j^{(h)} = 1. \end{cases}$

From the performance viewpoint, we note that each step is executed by a circuit with size $O(N)$ in constant depth. This is trivial for Steps (1), (2), (4), (5), and (6); as for Step (3), this result is achieved by the prefix-OR circuit of [CFL83b]. Thus, the Boolean circuit implementing the above computation has linear size and constant depth.

We now turn our attention to the correctness of the above procedure. We assume inductively that $e^{(h-1)}$ and $y^{(h-1)}$ are correct. The basis for the induction is provided by (1). Since, by Step (3), $e^{(h)}$ is the PREFIX-OR of vector $u^{(h)}$, we wish to show that $u_j^{(h)} = 0$ for $j \leqq j_h$ and $u_{j_h+1}^{(h)} = 1$. Indeed, for $j \leqq j_{h-1}$ we have $u_j^{(h)} = 0$, since $e_j^{(h-1)} = 0$ by the inductive hypothesis. For $j = j_{h-1}+1, \cdots, j_h$, we argue as follows. Let $C^{(h-1)}$ and $C^{(h)}$ be the strongly connected components of the correct prefixes computed by stages $(h-1)$ and h, respectively. Depending upon whether $j_h \leqq j_{h-1} + m$ or not, we distinguish two cases:

(1) $j \leqq j_h \leqq j_{h-1} + m$. In this case, since $y_j^{(h-1)} = y_j$ for $j \leqq j_{h-1}$ (with $y_{j_{h-1}-1} \in C^{(h-1)}$ and $y_{j_{h-1}} \in C^{(h)}$), by virtue of Step (1) above, $z_j^{(h)} = y_j$ for $j \leqq j_{h-1} + m$. Thus the smallest value $i \in [j_{h-1}, j_{h-1} + m]$ for which $z_i^{(h)} \notin C^{(h)}$ is detected by the condition $(e_i^{(h-1)} = 1) \wedge (z_i^{(h)} \not\equiv y_i^{(h-1)})$ and $u_{i+1}^{(h)} = 1$ is correctly generated. Note that $u_j^{(h)} = 0$ for $j \leqq i$.

(2) $j_h \geqq j > j_{h-1} + m$. In this case, the precise value of y_{j-m+1} is not known, but we know that $y_{j-m+1} \in C^{(h)}$. Since w_j is the product of m semigroup elements, one of its prefixes is a recurrent element x of A. Then $y_{j-m+1} \cdot x$ is, by Theorem 2, a unique element of $C^{(h)}$, independent of the specific $y_{j-m+1} \in C^{(h)}$. From this point on, we can argue as in Case (1) above.

This analysis shows that Step (2) always generates $u_{j_h+1}^{(h)} = 1$, whereas $u_k^{(h)} = 0$ for $k \leqq j_h$. Therefore Step (3) correctly generates a vector $e^{(h)}$ such that $e_i^{(h)} = 0$ for $i \leqq j_h$ and $e_i^{(h)} = 1$ for $i > j_h$. Subsequently, Step (4) uniquely detects position j_h (since

$e_{j_h}^{(h)} e_{j_h+1}^{(h)} = 01$), and Steps (5) and (6) construct vector $y^{(h)}$ in a straighforward manner. This completes the correctness proof and we have Theorem 3.

THEOREM 3. *If $\langle A, \cdot \rangle$ is an* NMC*-semigroups, then there exists a length-N bounded-depth prefix circuits for $\langle A, \cdot \rangle$ of size $O(N)$.*

5. Remarks and conclusions. The results of [CFL83a] and [CFL83b] and of this paper show that the optimal size of constant-depth prefix circuits is exponential for nongroup-free semigroups, slightly superlinear for group-free MC semigroups, and linear for (group-free) NMC semigroups.

It is interesting to contrast these results with those on the size-time complexity of prefix Boolean networks [BP89]. In this context, semigroups can be divided into two classes: *friction* semigroups, for which the size-time product is superlinear ($ST = \Theta(N \log (N/T))$), and *frictionless* semigroups, for which the size-time product is linear ($ST = \Theta(N)$). The class of frction semigroups can be more precisely defined as the intersection of two other classes: The NCF semigroups and the memory-inducive semigroups. The latter are those in which products of arbitrary length are true functions of all their factors, and can be structurally characterized as those semigroups whose recurrent subsemigroup is not isomorphic to a direct product of a left-zero and a right-zero semigroup. The inclusion relationship among the relevant classes of semigroups mentioned above is illustrated by means of a Venn diagram in Fig. 2.

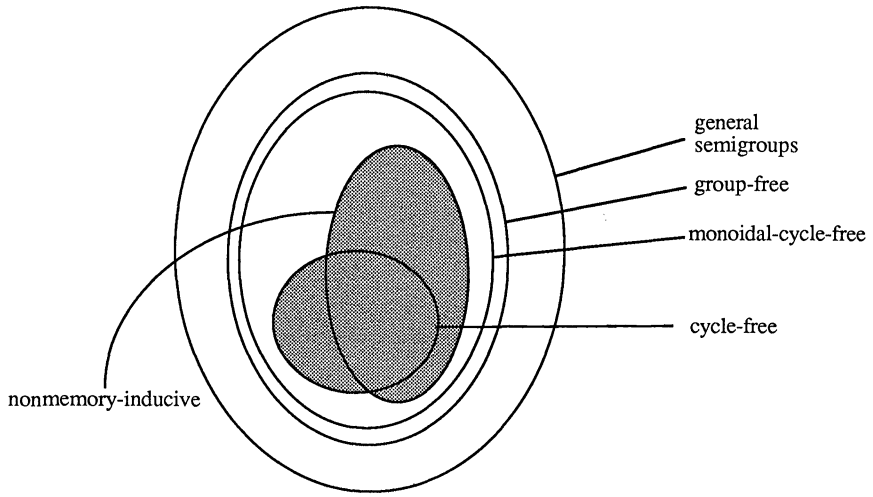


FIG. 2. *Inclusion relationships among classes of semigroups with respect to the complexity of their prefix circuits. The shaded area indicates the frictionless semigroups.*

We conclude by mentioning an open problem. We can show that cycle-free and right-zero semigroups are NMC. We can also show that the class of NMC semigroups is closed under direct product and under homomorphism. It would be interesting to ascertain whether NMC semigroups can be characterized as the homomorphic images of direct products of particular types of semigroups.

Acknowledgment. We are indebted to N. Pippenger for bringing to our attention the problem solved in this paper, for providing several penetrating suggestions, and for pointing out several relevant references.

REFERENCES

[BK82] R. P. BRENT AND H. T. KUNG, *A regular layout for parallel adders*, IEEE Trans. Comput., 31 (1982), pp. 260-264.

[BP85] G. BILARDI AND F. P. PREPARATA, *The influence of key length on the area-time complexity of sorting*, in Proc. ICALP, W. Brauer, ed., Nafplion, Greece, Springer-Verlag, Berlin, New York, 1985, pp. 53-62.

[BP86] ———, *Digital filtering in* VLSI, in Proc. Aegean Workshop on Computing, F. Makedon et al., eds., Loutraki, Greece, Springer-Verlag, Berlin, New York, 1986, pp. 1-11.

[BP89] ———, *Size-time complexity of boolean networks for prefix computations, J. Assoc. Comput. Mach.*, 36 (1989), pp. 362-382.

[CFL83a] A. K. CHANDRA, S. FORTUNE, AND R. LIPTON, *Unbounded fan-in circuits and associative functions*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, Association for Computing Machinery, New York, 1983, pp. 52-60.

[CFL83b] ———, *Lower bounds for constant depth circuits for prefix problems*, in Proc. ICALP, 1983.

[CP61] A. H. CLIFFORD AND G. B. PRESTON, *The Algebraic Theory of Semigroups*, Vol. 1. American Mathematical Society, Providence, RI, 1961.

[CS85] R. COLE AND A. SIEGEL, *On information flow and sorting: New upper and lower bounds for* VLSI *circuits*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, Portland, OR, IEEE Computer Society, Washington, DC, 1985, pp. 208-221.

[CV86] R. COLE AND U. VISHKIN, *Approximate and exact parallel scheduling with applications to list, tree and graph problems*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Canada, IEEE Computer Society, WAshington, DC, 1986, pp. 478-491.

[DS83] E. DEKEL AND S. SAHNI, *Binary trees and parallel scheduling algorithms*, IEEE Trans. Comput., 32 (1982), pp. 307-315.

[DDPW83] D. DOLEV, C. DWORK, N. PIPPENGER, AND A. WIDGERSON, *Superconcentrators, generalizers, and generalized connectors with limited depth*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, Association for Computing Machinery, New York, 1983, pp. 42-51.

[F83] F. E. FITCH, *New bounds for parallel prefix circuits*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, Association for Computing Machinery, New York, 1983, pp. 100-109.

[FSS81] M. FURST, J. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial time hierarchy*, in Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1981, pp. 260-270.

[H86] J. HASTAD, *Almost optimal lower bounds for small depth circuits*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Berkeley, CA, Association for Computing Machinery, New York, 1986, pp. 6-20.

[Hl85] W. D. HILLIS, *The Connection Machine*, MIT Press, Cambridge MA, 1985.

[GGKMRS83] A. GOTTLIEB, R. GRISHMAN, C. P. KRUSKAL, K. P. MCAULIFFE, L. RUDOLPH, AND M. SMIR, *The* NYU *Ultracomputer—Designing an* MIMD *shared memory parallel machine*, IEEE Trans. Comput., 32 (1983), pp. 175-189.

[K78] D. J. KUCK, *The Structure of Computers and Computations*, John Wiley, New York, 1978.

[KRS85] C. P. KRUSKAL, L. RUDOLPH, AND M. SNIR, *The power of parallel prefix*, IEEE Trans. Comput., 34 (1985), pp. 965-968.

[LF80] R. E. LADNER AND M. J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831-838.

[MP71] R. MCNAUGHTON AND S. PAPERT, *Counter-free Automata*, MIT Press, Cambridge, MA, 1971.

[O63] YU. OFMAN, *On the algorithmic complexity of discrete functions*, Soviet Phys. Dokl., 7 (1963), pp. 589-591.

[P87] N. PIPPENGER, *The complexity of computations by networks*, IBM J. Res. Develop., 23 (1987), pp. 235-243.

[P] ———, private communication, 1989.

[R84] J. REIF, *Probabilistic parallel prefix computation*, in Proc. 1984 Internat. Conference on Parallel Processing, August 1984, pp. 291-298.

[Rn89] A. G. RANADE, *Fluent Parallel Computation*, Ph.D. thesis, Department of Computer Science, Yale University, May 1989.

[S86] M. SNIR, *Depth-size trade-offs for parallel prefix computation*, J. Algorithms, 7 (1986), pp. 185-201.

[V76] L. G. VALIANT, *Graph-theoretic properties in computational complexity*, J. Comput. Systems Sci., 13 (1976), pp. 278-285.

[Y85] A. C. YAO, *Separating the polynomial-time hierarchy by oracles*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, Portland, OR, IEEE Computer Society, Washington, DC, 1985, pp. 1-10.

[Z70] Y. ZALCSTEIN, *On star-free events*, Proc. 11th Annual Symposium on Switching and Automata Theory, IEEE, NY, pp. 76-80.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 256-266, April 1990

004

A TIME-RANDOMNESS TRADE-OFF FOR OBLIVIOUS ROUTING*

DAVID PELEG† AND ELI UPFAL‡

Abstract. Three parameters characterize the performance of a probabilistic algorithm: T, the run-time of the algorithm; Q, the probability that the algorithm fails to complete the computation in the first T steps; and R, the amount of randomness used by the algorithm, measured by the entropy of its random source.

A tight trade-off between these three parameters for the problem of oblivious packet routing on N-vertex bounded-degree networks is presented. A $(1-Q)\log(N/T)-\log Q-O(1)$ lower bound for the entropy of a random source of any oblivious packet routing algorithm that routes an arbitrary permutation in T steps with probability $1-Q$ is proved. It is shown that this lower bound is almost optimal by proving the existence, for every $e^3 \log N \leqq T \leqq N^{1/2}$, of an oblivious algorithm that terminates in T steps with probability $1-Q$ and uses $(1-Q+o(1))\log(N/T)-\log Q$ independent random bits.

This result is complemented with an explicit construction of a family of oblivious algorithms that use less than a factor of $\log N$ more random bits than the optimal algorithm achieving the same run-time.

Key words. communication networks, permutation routing, randomized algorithms, resource trade-off

AMS(MOS) subject classifications. 68M10, 68Q25, 68R05

1. Introduction.

The contribution of randomness to computation is one of the least understood aspects of complexity theory. While there is a variety of problems for which probabilistic algorithms perform significantly better than the best known deterministic algorithms, we lack a clear theory that explains this phenomenon.

One approach that might extend our understanding of randomness is to treat randomness as a resource and to analyze the amount of randomness required in order to achieve a given performance. Since different algorithms might use different (biased) sources of randomness we need a uniform measure for the amount of randomness provided by different random sources. The most general measure for randomness is the entropy of the random source. Knuth and Yao [KY] have shown that this measure is closely related to a more algorithmically oriented measure, namely, the average complexity of simulating the random source using only independent random bits as a source of randomness. For algorithms that use only independent random bits we distinguish between two measures of randomness: the *average* number of random bits used by the algorithm (or alternatively the entropy); and the *worst case* number of random bits. We prove a gap between these two measures.

Our goal is to study the trade-off between the amount of randomness used by an algorithm and its performance, measured either by the probability that it fails to complete the computation within a given number of steps, or by its average run time. In order to prove such a trade-off between randomness and performance for a given problem, we need to prove a gap between the deterministic and probabilistic complexity

* Received by the editors November 16, 1987; accepted for publication June 21, 1989. A preliminary version of this work was presented at the Twentieth Annual ACM-SIGACT Symposium on Theory of Computing, Chicago, IL, May, 1988.

† Computer Science Department, Stanford University, Stanford, California 94305. Present address, Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel. This work was partially supported by a Weizmann fellowship and by Office of Naval Research contract N00014-85-C-0731.

‡ IBM Almaden Research Center, 650 Harry Road, San Jose, California 95120, and Department of Applied Mathematics, The Weizmann Institute of Science, Rehovot 76100, Israel. The work of this author while at the Weizmann Institute was partially supported by a Bat-Sheva de Rothschild Award, and by a Revson Career Development Award.

of the problem. Since in general it is not known whether probabilistic time is different from deterministic time, previous works mainly consider the question of reducing the number of random bits used to solve a given problem [B], [CG], [KR].

In this work we focus on one of the few problems whose probabilistic complexity is provably better than its deterministic one, namely, the problem of oblivious routing in bounded-degree networks. A routing algorithm is *oblivious* if the route of one packet does not depend on the origins and destinations of other packets in the system. Let N denote the number of processors in the system. In each step each processor can send up to one packet to each direct neighbor in the network. We measure the performance of a routing algorithm by the time it requires to execute a *permutation request*, i.e., a communication request of N packets with initially one packet at each processor, and where each processor appears as a destination of exactly one packet.

Borodin and Hopcroft [BH] have shown that any deterministic oblivious packet routing algorithm on a bounded-degree network requires $\Omega(\sqrt{N})$ parallel steps.[1] Aleliunas [A] and Upfal [Up] have presented probabilistic oblivious routing algorithms for several bounded-degree networks that terminate in $O(\log N)$ parallel time.

In this work we bridge the gap between these two results by analyzing the minimum amount of randomness needed in order to execute oblivious routing on a bounded-degree network in T steps for any $\log N \leqq T < O(\sqrt{N})$. We first extend the lower bound of [BH] to show that a probabilistic algorithm that uses a random source with entropy bounded by $\frac{1}{2}\log N$ cannot run faster than the $O(\sqrt{N})$ deterministic algorithm.[2] For larger amounts of randomness we prove a tight trade-off between the entropy of the random source of the algorithm (or the worst-case number of random bits used by the algorithm), its run-time, and its failure probability. In particular, we show that each additional random bit can reduce the product of the run-time and the failure probability by a factor of $\frac{1}{2}$.

THEOREM 1. *Let* $\log N \leqq T \leqq \sqrt{N}/2\sqrt{d}$, *and consider an oblivious routing algorithm that terminates in* T *steps with probability* $1-Q$ *on an* N*-vertex degree* d *network.*

(1) *The algorithm must use a random source with entropy at least* $(1-Q)\log(N/T) - \log Q - O(1)$.

(2) *If the algorithm uses only independent random bits, then it must use at least* $\log(N/T) - \log Q - O(1)$ *random bits in the worst case.* □

Since the entropy of the distribution defined by the outcome of k independent random bits is equal to k, the following theorem shows that the above lower bound is almost optimal.

THEOREM 2. *For every* $e^3 \log N \leqq T \leqq \sqrt{N}$ *there exists an oblivious algorithm on the butterfly network (degree* 4*) that terminates in* T *steps with probability* $1-Q$ *and uses* $(1-Q+o(1))\log(N/T) - \log Q$ *independent random bits on the average and no more than* $(1+o(1))\log(N/T) - \log Q$ *independent random bits in the worst case.* □

Similar results are obtained for the trade-off between randomness and average run-time of oblivious routing algorithms. Note that $\Omega(\log N)$ is a lower bound for the execution time of any routing algorithm, and there is a deterministic oblivious routing algorithm on a bounded degree network that terminates in $O(\sqrt{N})$ parallel steps [L].

Our tight upper bound is not constructive, since it involves a structure that we cannot explicitly construct in an efficient way. Using a family of universal hash functions

[1] Our definition of an oblivious algorithm is slightly weaker than the definition in [BH], thus resulting in a stronger model of computation. However, the lower bound in [BH] holds even in the stronger model.

[2] All logarithms in this paper are taken to base 2.

[CW], we can obtain a simple and easy-to-program family of oblivious routing algorithms. The number of random bits used by these algorithms is optimal (up to a constant factor) for $T = N^{\varepsilon}$ for some $\varepsilon > 0$, and less than a factor of $\log N$ from the optimal for $T = o(N^{\varepsilon})$. The case $T = O(\log N)$ and $R = O(\log^2 N)$ was first proven in [R] using standard hashing techniques. For smaller R (and larger T) we need a new "divide-and-conquer" proof technique that yields better probabilistic estimates when using smaller families of hash functions. Oblivious routing on the hypercube using simpler randomization schemes has been analyzed in [KR]. For every $1 \leqq f(N) \leqq \log N$, Karloff and Raghavan present a scheme that runs in $f(N)N^{(c+1)/f(N)} \log N$ time and uses $f(N) \log N$ independent random bits.

While our interest in oblivious routing on bounded-degree network was motivated by theoretical questions, we should note that oblivious routing is, in fact, a natural and practical problem to study. Technical considerations dictate the use of oblivious routing algorithms in almost all practical implementations of large scale communication networks. The fact that very limited randomness is sufficient for efficient oblivious routing might offer some explanation as to why some of the simple heuristics for oblivious routing perform so well in practice.

2. Definitions. We consider a collection of N processors connected by some bounded-degree communication network. The processors generate packets that have to be transmitted through the network to their destinations. In each step each processor can transmit one packet to one of its direct neighbors in the network. Every packet contains a label with its destination, thus without loss of generality we can assume that the number of bits a processor can transmit to a direct neighbor in one step is at least $\log N$.

In this paper we are interested in probabilistic oblivious routing algorithms.

DEFINITION 2.1. A (probabilistic) algorithm is oblivious if the route taken by one packet does not depend on the source or destination of any other packet in the network.

We measure the performance of a routing algorithm by the time it requires to execute any permutation request. Given a permutation $\sigma \in \Sigma_N$ on N elements, a *permutation request* σ is a set of N packets, initially located one at each processor, such that for every $1 \leqq i \leqq N$ the destination of the packet originated at processor i is $\sigma(i)$.

A probabilistic algorithm might use a biased source of randomness. We use the entropy function as a uniform measure for the amount of randomness used by the algorithm.

DEFINITION 2.2. The *entropy function* of the probability distribution $\bar{P} = \{p_1, p_2, \cdots\}$ *is* $H(\bar{P}) = \Sigma_i - p_i \log p_i$.

The entropy function is closely related to a more algorithmically oriented measure of randomness defined by Knuth and Yao [KY]. Let A be an algorithm that generates the distribution $\bar{P} = \{p_1, p_2, \cdots\}$ using only independent random bits as its random source. Let $R(A)$ denote the average execution time of the algorithm A, and let $\tilde{R}(\bar{P}) = \min_A R(A)$.

THEOREM 2.1 [KY]. $H(\bar{P}) \leqq \tilde{R}(\bar{P}) \leqq H(\bar{P}) + 2$. □

The algorithm can generate no more than one random bit per step. Thus, up to a constant additive factor, the entropy of a random source specifies the average number of independent random bits required to simulate the source.

3. The lower bound. In this section we consider an N-vertex network $G = (V, E)$ with maximal in-degree d for which a probabilistic oblivious routing algorithm is sought. The lower bound extends that of [BH] in a natural way.

LEMMA 3.1. *Let* $\log N \leqq T \leqq \sqrt{N}/2\sqrt{d}$. *For any collection* $\{D_1, \cdots, D_l\}$ *of* $l \leqq (1/4\sqrt{d})(N/T)$ *deterministic oblivious routing algorithms there exists an input permutation* σ *such that each of these algorithms requires at least* T *steps on this input.*

Proof. In a deterministic oblivious routing algorithm, the route taken by a packet is completely determined by its source and destination. Any such algorithm can therefore be described in full by means of a collection of N^2 routes $\rho(w, v)$, one for every (ordered) pair of vertices in the network. If the route $\rho(w, u)$ goes through a vertex x then we call w *an origin* for x.

A *partial destination graph* $G(W, u)$ is a collection of routes originated at the vertices of W and directed to u, i.e., $G(W, u) = \{\rho(w, u) \mid w \in W\}$. For every vertex $x \in V$, denote by $org(x, G(W, u))$ the number of origins x has in $G(W, u)$. For every $k \geqq 1$ let $S_k(G(W, u)) = \{x \mid x \in V, org(x, G(W, u)) \geqq k\}$. A *full destination graph* $G(u)$ is the partial destination graph $G(V, u)$ (we henceforth omit the words partial/full whenever no confusion may arise).

We need the following generalization of Lemma 1 in [BH].

LEMMA 3.1.1. *For every destination graph* $G(W, u)$ *and for every* k *such that* $1 \leqq k \leqq |W|$, $|S_k(G(W, u))| \geqq |W|/(dk - d + 1)$.

Proof. The proof is a simple variation of that of [BH]. Let $S = S_k(G(W, u))$. Since $u \in S$, every route in $G(W, u)$ enters S at some point. Let P denote the set of routes in $G(W, u)$ originated in $V \backslash S$. Then $|P| + |S| \geqq |W|$. We bound $|P|$ from above by counting the number of times that a route from P enters S for the first time. Let $H = \Gamma(S) \backslash S$. Since the graph G has maximal degree d, $|H| \leqq d|S|$. Every route in P must enter S through H. However, at most $k-1$ routes can go through every vertex $x \in H$ (otherwise $org(x) \geqq k$ and hence $x \in S$). Therefore $|P| \leqq (k-1)|H| \leqq (k-1)d|S|$, and $(dk - d + 1)|S| \geqq |W|$. □

Let T be given and consider a collection of algorithms as above. A deterministic oblivious routing algorithm D can be described in terms of a collection of N full destination graphs, one for each vertex in the network. Consequently, denote $D_i = \{G_i(u) \mid u \in V\}$, for every $1 \leqq i \leqq l$. We now describe how to construct an input permutation σ that will force each of these algorithms to run for a long time. The permutation σ is built iteratively, similar to the construction of [BH]. For each D_i we fix exactly T source-destination pairs in σ.

Suppose we already chose a partial permutation $\sigma = \{(s_j, t_j) \mid 1 \leqq j \leqq (i-1)T\}$ ($\sigma = \varnothing$ for $i = 1$) taking care of D_1 through D_{i-1} ($1 \leqq i \leqq l$) and consider D_i. Eliminate from D_i all destination graphs $G(t_j)$, $1 \leqq j \leqq (i-1)T$. From every other destination graph $G_i(u)$ in D_i eliminate the routes $\rho(s_j, u)$, $1 \leqq j \leqq (i-1)T$. This leaves us with a collection of $m = n - (i-1)T$ partial destination graphs $\{G_i'(W_1, u) \mid u \in W_2\}$, where $W_1 = V \backslash \{s_j \mid 1 \leqq j \leqq (i-1)T\}$ and $W_2 = V \backslash \{t_j \mid 1 \leqq j \leqq (i-1)T\}$, $|W_1| = |W_2| = m$. By the restrictions on l and T,

$$(*) \qquad m \geqq N - lT \geqq \frac{N}{2}.$$

By Lemma 3.1.1, $|S_T(G_i'(W_1, u))| \geqq m/(dT)$ for every $u \in W_2$. For every vertex x let c_x denote the number of vertices $u \in W_2$ such that $x \in S_T(G_i'(W_1, u))$. Then

$$\sum_{x \in V} c_x = \sum_{u \in W_2} |S_T(G_i'(W_1, u))| \geqq \frac{m^2}{dT}.$$

Therefore there must be a vertex x such that $c_x \geqq m^2/dTN$. By (*), $m^2 \geqq N^2/4 \geqq dNT^2$, so $c_x \geqq T$. We now select precisely T pairs of distinct vertices (w_j, u_j), $1 \leqq j \leqq T$, such that for each j, $w_j \in W_1$, $u_j \in W_2$, $x \in S_T(G_i'(W_1, u_j))$ and w_j is an origin of x in

$G_i'(W_1, u_j)$, and add these pairs to the permutation σ. By the choice of W_1 and W_2, these vertices are also distinct from all previously chosen vertices in σ. Any run of the algorithm D_i on an input permutation containing σ as a partial permutation requires the packets sent between the endpoints of these pairs to go through the vertex x, and at most one of these T packets can be sent out of x in each time unit, so the number of steps is at least T. (Of course, $\Omega(\log n)$ is also a lower bound.) □

THEOREM 3.2. *Let* $\log N \leqq T \leqq \sqrt{N}/2\sqrt{d}$ *and* $0 < Q < 1$, *and let* $\mathcal{A}$ *be an oblivious algorithm that terminates on every input in* T *steps with probability* $1-Q$.

(1) *Algorithm* $\mathcal{A}$ *must use a random source with entropy at least* $(1-Q)\log(N/T) - \log Q - \log(5\sqrt{d})$.

(2) *If* $\mathcal{A}$ *uses independent random bits as its source of randomness then the number of random bits* $\mathcal{A}$ *uses in the worst case is at least* $\log(N/T) - \log Q - \log(5\sqrt{d})$.

Proof. Any probabilistic algorithm $\mathcal{A}$ can be transformed to an equivalent form in which the random source of $\mathcal{A}$ first chooses a deterministic algorithm from a pre-defined collection of deterministic algorithms $\mathcal{B} = \{B_1, B_2, \cdots\}$ and then the selected algorithm is deterministically executed. Let $\tilde{\mathcal{A}}$ be a routing algorithm in this form, which is equivalent to the algorithm $\mathcal{A}$. Let $\bar{P} = \{p_1, p_2, \cdots\}$ be the probability distribution by which the deterministic algorithms of $\mathcal{B}$ are selected and assume that $p_1 \geqq p_2 \geqq \cdots$. Since $\mathcal{A}$ is an oblivious algorithm, the deterministic algorithms $B_1, B_2, \cdots$ are also oblivious. By the previous lemma, for any set of $l = \lfloor (1/4\sqrt{d})(N/T) \rfloor$ deterministic algorithms there is an input permutation σ such that each of these deterministic algorithms requires more than T steps on this input. If the probabilistic algorithm terminates on every input with probability $1-Q$ in T steps, the sum of the probabilities given to any set of l deterministic algorithms must be bounded by Q. In particular, $W = \sum_{i=1}^{l} p_i \leqq Q$, thus for every $i \geqq l$, $p_i \leqq Q/l$. Therefore,

$$\sum_{i \geqq l+1} -p_i \log p_i \geqq \sum_{i \geqq l+1} -p_i \log \frac{Q}{l} = (1-W)\log \frac{l}{Q}.$$

Similarly, since $p_i \leqq Q$ for every i,

$$\sum_{1 \leqq i \leqq l} -p_i \log p_i \geqq \sum_{1 \leqq i \leqq l} -p_i \log Q = W \log \frac{1}{Q}.$$

Put together, we get that

$$\sum_i -p_i \log p_i \geqq (1-Q)\log l - \log Q \geqq (1-Q)\log \frac{N}{T} - \log Q - \log(5\sqrt{d}),$$

which proves part (1).

If the sum of every subset of l probabilities is bounded by Q then there must be at least l/Q determinstic algorithms that are selected with positive probability. Thus, at least one of the probabilities is Q/l or less, which requires at least $-\log(Q/l) = \log(N/T) - \log Q - \log(5\sqrt{d})$ bits to generate. □

THEOREM 3.3. *Let* $\log N \leqq T \leqq \sqrt{N}/2\sqrt{d}$, *and let* $\mathcal{A}$ *be an oblivious algorithm that terminates on every input in average time* T.

(1) *Algorithm* $\mathcal{A}$ *must use a random source with entropy at least* $(1 - T/\sqrt{N})\log\sqrt{N} - \log(T/\sqrt{N}) - (5\sqrt{d})$.

(2) *If* $\mathcal{A}$ *uses independent random bits as its source of randomness, then the number of random bits* $\mathcal{A}$ *uses in the worst case is at least* $\log(N/T) - (5\sqrt{d})$.

Proof. If the algorithm terminates in T steps on the average then the probability that it terminates in no more than $\sqrt{N}$ steps must be at most $T/\sqrt{N}$. □

4. The upper bound.

4.1. The general scheme. In this section we present algorithms for the butterfly network of degree 4. Similar algorithms can be derived for related topologies such as the hypercube, the omega network, and the cube-connected-cycles network (cf. [Ul]).

For simplicity, we assume that $N = n2^n$ for some $n \geqq 1$. To construct the butterfly network we separate the processors' numbers, in their binary representation, into two parts. The n rightmost bits of the number x are called the *address*, denoted by *address*(x); the rest is called the *prefix* and is denoted by *prefix*(x). We sometimes describe a processor's name as the pair (*prefix*(x), *address*(x)). Each group of 2^n processors having the same prefix α forms one stage in the network. Each processor with prefix α has four neighbors, two with prefix $(\alpha+1) \bmod n$ and two with prefix $(\alpha-1) \bmod n$. Specifically, the processor $(\alpha, (b_0, \cdots, b_\alpha, \cdots, b_{n-1}))$ is connected to the processors with the addresses $b_0, \cdots, b_\alpha, \cdots, b_{n-1}$ and $b_0, \cdots, \bar{b}_\alpha, \cdots, b_{n-1}$ and prefix $(\alpha+1) \bmod n$; and to the processors with address $b_0, \cdots, b_\alpha, \cdots, b_{n-1}$ and $b_0, \cdots, \bar{b}_{(\alpha-1) \bmod n}, \cdots, b_{n-1}$ and prefix $(\alpha-1) \bmod n$.

All the known probabilistic packet routing algorithms are based on the following two phase routing scheme first presented by Valiant [V]:

(1) For each packet x choose a random destination $\rho(x)$.

(2) Send each packet x to its random destination $\rho(x)$;

(3) Send the packets from their random places to their real destinations;

Our model of computation allows each processor to send exactly one packet per step. Each processor keeps priority queues of packets waiting for transmission in each edge. The priority of a message is the number of edges it still has to pass in the current phase. Thus, the priority numbers are used to speed up slower messages at the expense of faster ones.

It has been proven in [A] and [Up] that this algorithm terminates with high probability in $O(\log N)$ parallel steps. The algorithm requires $O(N \log N)$ random bits since N random destinations have to be chosen independently for the N packets.

We reduce the number of random bits used by the algorithm by choosing the middle destinations of the N packets from a smaller family of possible sets of destinations. To handle the dependencies between the paths of the different packets we split the algorithm into three routing phases. In the algorithm we distinguish between forward and backward paths. There are two paths connecting every two processors in the same stage. The *forward* path is the path that traverses the stages in increasing order, while the *backward* path traverses the stages in decreasing order.

Let $\mathcal{AS}$ be a set of L possible *intermediate assignments*, i.e., functions $\{f_1, \cdots, f_L\}$, $f_i : [1, \cdots, N] \to [1, \cdots, N]$, and let $\mathcal{P}(\mathcal{AS})$ be a probability distribution on $\mathcal{AS}$.

ALGORITHM ROUTE.

Step 1. Processor 1 randomly chooses a function f_i from $\mathcal{AS}$ according to the distribution $\mathcal{P}(\mathcal{AS})$.

Step 2. Broadcast f_i to all processors in the network.

Step 3. For each message with destination x initiated at a processor v, v computes $f_i(x) = (\textit{prefix}(f_i(x)), \textit{address}(f_i(x)))$, the prefix and address of the intermediate assignment of x.

Step 4. *First routing phase.* Each message with destination x that was initiated at a processor $v = (\alpha, \beta)$ is sent to the processor $(\alpha, \textit{address}(f_i(x)))$ using the forward path.

Step 5. *Second routing phase.* A message with destination x currently at processor $(\alpha, \textit{address}(f_i(x)))$ is sent to the processor $(\alpha, \textit{address}(x))$ using the backward path.

Step 6. *Third routing phase.* A message currently at the processor $(\alpha, address(x))$ is sent to its final destination $x = (prefix(x), address(x))$ using only the direct edges connecting processors with the same address.

The efficiency of the algorithm depends on the choice of the family $\mathcal{AS}$ and the distribution $\mathcal{P}(\mathcal{AS})$. The rest of this section is concerned with the characterization and construction of good families for intermediate assignments.

We bound the execution time of each communication phase using the *delay sequence* technique [Up]. Let $\{P_1, \cdots, P_N\}$ be a set of paths that the N packets have to traverse in one of the communication phases. A *delay sequence* D is a sequence of processors $(v_n, \cdots, v_0)$ such that for any $i > 1$ either $v_i = v_{i-1}$ or v_i is one of the two processors transmitting packets to v_{i-1} in that communication phase.

For a given delay sequence D and a set of paths $\{P_1, \cdots, P_N\}$ let f_i^D denote the number of messages with priority i leaving processor v_i, and let $F^D = \sum_{i=1}^{n} f_i^D$. F^D is called the *volume* of the delay sequence D.

LEMMA 4.1 [Up]. *Let T denote the number of parallel steps required to execute a phase with the set of paths $\{P_1, \cdots, P_N\}$, then $T \leqq \max_D F^D$.* □

4.2. Oblivious routing with fewer random bits.

DEFINITION 4.1. A function $f, f:[1, \cdots, N] \to [1, \cdots, N]$ is an (N, T)-*mixer* for a permutation $\sigma \in \Sigma_N$ if the execution of the first and second routing phases with input σ and intermediate addresses given by f terminates in T steps.

Our goal is to construct a small set of functions, such that for every permutation σ most of the functions of the set are (N, T)-mixers for σ. It has been shown in [Up] that for every $\sigma \in \Sigma_N$, and $T = O(\log N)$, if f is a random permutation in Σ_N, then with high probability f is an (N, T)-mixer for σ. Using a counting argument we will show that there exists a small subset of Σ_N, such that for every permutation $\sigma \in \Sigma_N$, most of the functions in the subset are (N, T)-mixers for σ.

LEMMA 4.2. *Let $e^3 \log N \leqq T \leqq \sqrt{N}$, and let f be a permutation chosen randomly with uniform distribution from the set of all permutations on N elements. Let $\sigma \in \Sigma_N$; then*

$$\text{Prob}\{f \text{ is an } (N, T)\text{-mixer for } \sigma\} \geqq 1 - e^{-T}.$$

Proof. We first bound the probability that in the execution of the first routing phase with input σ and the function f there will be a delay sequence with volume greater than t. We use the following facts:

(1) There are no more than $N3^n$ different delay sequences.

(2) If the volume of a delay sequence is t, it can be partitioned between the n vertices of the sequence in no more than $\binom{t+n+1}{n}$ ways.

(3) The t_i elements that leave vertex v_i with priority i can be originated in 2^{n-i} vertices and their destinations can be chosen from $n2^i$ possible locations.

Thus, the probability is bounded by

$$\begin{aligned} & N3^n \binom{t+n+1}{n} \frac{(N-t)!}{N!} \prod_{i=1}^{n} \binom{2^{n-i}}{t_i} \binom{n2^i}{t_i} \\ & \leqq N3^n \left(\frac{(t+n+1)e}{n}\right)^n \left(\frac{1}{(N-t)}\right)^t 2^{nt} e^{2t} n^t \prod_{i=1}^{n} \frac{1}{t_i^{2t_i}} \\ & \leqq N3^n e^{t+1} e^n \left(\frac{1}{n2^n}\right)^t \left(1+\frac{t}{N}\right)^t 2^{nt} e^{2t} n^t \left(\frac{n}{t}\right)^{2t} \leqq \frac{1}{2} e^{-2t}, \end{aligned}$$

for $t \geqq e^3 n$.

The number of packets leaving processor v_i with priority i in the execution of the second routing phase is equal to the number of packets leaving processor v_i with priority $n-i$ in the execution of the first phase when it runs with the identity permutation as input and with the same intermediate function. Thus, the probability of having a delay sequence with volume t in the execution of the second routing phase with input σ and the function f is also bounded by $\frac{1}{2}e^{-2t}$, and the probability that f is not an (N, T)-mixer is bounded by $1-e^{-T}$. □

LEMMA 4.3. *For every* $e^3 \log N \leqq T \leqq \sqrt{N}$, $e^{-T} \leqq Q \leqq \frac{1}{2}$ *and* $L \leqq \max\{16N \log N/QT, 16N(\log n)^2/T\}$ *there exists a set* $\mathscr{AS}$ *of* L *functions such that for every pemutation* $\sigma \in \Sigma_N$ *at most* $4N \log N/T$ *of the functions in* $\mathscr{AS}$ *are not* (N, T)*-mixers for* σ.

Proof. Let $\mathscr{RAS}$ be a random set of L permutations chosen with uniform probability from the set of all permutations on N elements. We show that with positive probability the set $\mathscr{RAS}$ has the required property.

By Lemma 4.2, for every permutation σ if $f \in \mathscr{RAS}$ then f is an (N, T)-mixer for σ with probability $1-e^{-T}$. The probability that $L(T) = 4N \log N/T$ functions in $\mathscr{RAS}$ are not (N, T)-mixers for σ is bounded by $\binom{L}{L(T)} e^{-TL(T)}$ and the probability that for any permutation σ less than $L-L(T)$ functions are (N, T)-mixers for σ is bounded by $N!2^L e^{-TL(T)}$. For the case $Q \geqq 1/\log N$ we get

$$N!(4e \log N)^{N \log N/T} e^{-4N \log N} < 1.$$

For $Q \leqq 1/\log N$ we get

$$N!\left(\frac{4e}{Q}\right)^{N \log N/T} e^{-4N \log N} < 1.$$

Thus, there exists a set $\mathscr{AS}$ as required. □

THEOREM 4.4. *For every* $e^3 \log N \leqq T \leqq \sqrt{N}$ *and* $e^{-T/2} \leqq Q \leqq \frac{1}{2}$ *there exists an oblivous routing algorithm that uses a random source with entropy* $(1-Q+o(1)) \log (N/T) - \log Q$ *and terminates in* T *steps with probability at least* $1-Q$.

Proof. We use the algorithm ROUTE with a set $\mathscr{AS}$ of $L \leqq \max\{32N \log N/QT, 32N(\log n)^2/T\}$ satisfying the requirement of Lemma 4.3. We define the following distribution $\mathscr{P}(\mathscr{AS})$ on the set $\mathscr{AS}$:

$$p_i = \begin{cases} \dfrac{Q \log N}{(\log N+1)}, & i=1 \text{ and } Q > \dfrac{1}{\log N}, \\ \dfrac{Q}{2}, & i=1 \text{ and } Q \leqq \dfrac{1}{\log N}, \\ \dfrac{1-p_1}{L-1}, & 2 \leqq i \leqq L. \end{cases}$$

The entropy E of the distribution is bounded by

$$\begin{aligned} E &\leqq -p_1 \log p_1 - (1-p_1) \log \frac{1-p_1}{L-1} \leqq -\log p_1 + \left(1-Q+\frac{1}{\log N}\right) \log 2p_1 L \\ &\leqq -\log Q + \left(1-Q+\frac{1}{\log N}\right) \log LQ + 2 \log \frac{2 \log N}{\log N+1} \\ &\leqq (1-Q+o(1)) \log \frac{N}{T} - \log Q. \end{aligned}$$

Since $\log L \leqq 2 \log N$, broadcasting a name of a function in $\mathscr{AS}$ requires no more than $2 \log N$ bits. Also, the diameter of the graph is $2 \log N$, so the broadcasting phase takes at most $2 \log N$ parallel steps.

The sum of every subset of $8N \log N/T$ probabilities in $\mathscr{P}(\mathscr{AS})$ is bounded by Q. Thus, with probability $1-Q$ the first and the second communication phases take no more than $T/4$ steps.

In the third routing phase, messages are sent along rings of n processors. Each processor is the distination of one message and the priority number of a message at a given processor is the distance of this processor from its destination. Thus only one message can leave each processor with a given priority and the maximum volume of a delay sequence at this routing phase is $n \leqq \log N$. Thus, the total run time of the algorithm is $T/4+3 \log N \leqq T$ with probability $1-Q$. □

Theorem 2.1 gives us a tool for simulating the distribution $\mathscr{P}(\mathscr{AS})$ using independent random bits.

COROLLARY 4.5. *For every* $e^3 \log N \leqq T \leqq \sqrt{N}$ *and* $e^{-T/2} \leqq Q \leqq \frac{1}{2}$ *there exists an oblivious routing algorithm that uses* $(1-Q+o(1)) \log (N/T) - \log Q$ *random bits on the average, and terminates in* T *steps with probability* $1-Q$.

Proof. By Theorem 2.1 the average number of independent random bits required to generate the distribution used in the proof of Theorem 4.4 is equal to the entropy of the distribution which is bounded by $\log N - \log Q$. □

Moreover, it is possible to match the lower bound of Theorem 3.3(2), for the *worst case* number of random bits used by the algorithm.

THEOREM 4.6. *For every* $e^3 \log N \leqq T \leqq \sqrt{N}$ *and* $e^{-T/4} \leqq Q \leqq \frac{1}{2}$ *there exists an oblivious routing algorithm that uses* $(1+o(1)) \log (N/T)$ *random bits in the worst case and terminates in* T *steps with probability* $1-Q$.

Proof. We use the algorithm ROUTE with a uniform distribution over a set $\mathscr{AS}$ of L functions such that:

(1) $\frac{1}{2} \cdot \max \{64N \log N/QT, 64N(\log n)^2/T\} \leqq L \leqq \max \{64N \log N/QT, 64N(\log n)^2/T\}$;

(2) For every $\sigma \in \Sigma_N$ no more than $16N \log N/T$ functions in $\mathscr{AS}$ are not $(N, T/4)$-mixers for σ;

(3) $L = 2^l$ for some integer l. The algorithm picks an element of $\mathscr{AS}$ with uniform distribution.

By Lemma 4.3 there exists such a set, and since $L \leqq \max \{64N \log N/QT, 64N(\log n)^2/T\}$, $l \leqq (1+o(1)) \log (N/QT)$ and this number bounds the worst-case number of random bits used by the algorithm.

Since the sum of every subset of $4N \log N/T$ probabilities is bounded by Q, the first and the second routing phases take $T/4$ with probability $1-Q$. The broadcasting phase takes at most $\max \{l, 2 \log N\}$ steps and the third routing phase takes $\log N$ steps. Thus, the algorithm terminates in $\log N - \log Q + T/4 + \log N \leqq T$ steps with probability $1-Q$. □

4.3. The average case. For the average case analysis of the algorithm we need a bound on its worst-case performance.

LEMMA 4.7. *Let* f *and* σ *be two permutations on* N *elements; then* f *is an* $(N, 4\sqrt{N \log N})$*-mixer for* σ.

Proof. For a given communication phase let c_o and c_i denote the bounds on the number of packets located in any one processor at the start and at the termination of the communication phase. The set of packets that leave processor i with priority i are originated in 2^i processors and can be sent to one of 2^{n-i} processors.

Thus, the volume of v_i, the ith element in any delay sequence is always bounded by $\min\{c_0 2^i, c_i 2^{n-i}\}$.

If f is a permutation then in the first phase $c_o = 1$ and $c_i = n$ and in the second phase $c_o = n$ and $c_i = 1$. Thus, any delay sequence in the two first routing phases is bounded by $2(n2^n)^{n/2} = 2\sqrt{nN}$ and f is an $(N, 4\sqrt{N \log N})$-mixer. □

THEOREM 4.8. *For every $e^3 \log N \leqq T \leqq \sqrt{N}$ there exist oblivious routing algorithms that use $(1+o(1))\log(N/T)$ independent random bits in the worst case and terminate in T steps on the average.* □

4.4. An explicit construction. We now turn to the question of explicit construction of an efficient intermediate assignments set. Let

$$\mathscr{EAS}(r) = \left\{ h \mid h(x) = \left(\left(\sum_{j=0}^{r} a_j x^j \right) \bmod P \right) \bmod N, \text{ for any } a_j \in [1, \cdots, P] \right\}$$

where $P \geqq N$ is a prime. Let $\mathscr{P}(\mathscr{AS})(r)$ be a uniform distribution on the $\mathscr{EAS}(r)$.

THEOREM 4.9. *Let $e^3 \log N \geqq T \geqq \sqrt{N}$ and let $r = (\log N - \log Q)/(\log T - \log n)$. If the algorithm* ROUTE *uses the pair $(\mathscr{EAS}(r), \mathscr{P}(\mathscr{AS})(r))$ then the algorithm terminates in T steps with probability $1-Q$.*

Proof. We first show that the first communication phase terminates in $T/4$ steps with probability $1 - Q/2$.

For each pair (v, i) let $A(v, i)$ denote the set of pairs of processors $\{x, y\}$ such that the forward path from x to y leaves processor v with priority i. Clearly, $|A(v, i)| = 2^n$. Let $A(v, i, 1), \cdots, A(v, i, T/4r)$ be a partition of $A(v, i)$ into $T/4r$ sets such that for every $1 \leqq j \leqq T/4r$, $|A(v, i, j)| \leqq 2r2^n/T + 1$. Let $A(j) = \bigcup_{v \in V} \bigcup_{0 \leqq i \leqq n} A(v, i, j)$.

Let F_j^D denote the contribution of communication between pairs in $A(j)$ to the volume of a delay sequence D.

If the first communication phase does not terminate on every input in $T/4$ steps with probability $1 - Q/2$ then there exists a permutation σ, a delay sequence D, and an index j, such that in the execution of the first routing phase with input σ

$$\text{Prob}\{F_j^D \geqq r\} \geqq \frac{Q}{2}.$$

To bound the probability of the above event we observe the following facts.

(1) There are no more than $N3^n$ different delay sequences.

(2) There are $T/4r$ sets $A(j)$.

(3) r can be divided in no more than $\binom{r+n+1}{r}$ ways between the n elements of the delay sequence.

(4) No more than $4rN/T$ pairs can contribute to the volume of each element in the delay sequence.

(5) $|\mathscr{EAS}(r)| = P^r$.

(6) Given a set of r pairs (x_l, y_l), $l = 1, \cdots, r$, no more than $(P/N)^r$ functions in $\mathscr{EAS}(r)$ satisfy the relation $y_l = f(x_l)$ for all $l = 1, \cdots, r$.

Using the above observations, we require

$$\text{Prob}\{F_j^D \geqq r\} \leqq N3^n \frac{T}{4r} \binom{n+r+1}{n} \left(\frac{4rN}{T}\right)^r \left(\frac{P}{N}\right)^r \left(\frac{1}{P}\right)^r \leqq \frac{Q}{2},$$

which implies

$$r \geqq \frac{\log N - \log Q}{\log T - \log n}.$$

As in the proof of Lemma 4.2 the same bound holds for the execution of the second communication phase. Following the proof of Theorem 4.4, execution of the first and the second communication phases in $T/2$ steps implies execution of the whole algorithm in T steps. □

COROLLARY 4.10. *The family of sets* $\mathcal{EAS}(r)$ *gives an explicit construction of oblivious routing algorithms that for every* $e^3 \log N \leqq T \leqq N^{1/2}$

(1) *Use* $\log N\ (\log N - \log Q)/(\log T - \log n)$ *random bits in the worst case and terminate in* T *steps with probability* $1 - Q$;

(2) *Use* $2 \log^2 N/(\log T - \log n)$ *random bits in the worst case and terminate in* T *steps on the average.* □

Acknowledgments. We wish to thank Eli Shamir for raising the question of packet routing with fewer random bits, and Allan Borodin, Nati Linial, and Nick Pippenger for many stimulating discussions and helpful ideas.

REFERENCES

[A] R. ALELIUNAS, *Randomized parallel communication*, in Proc. 1st ACM Symposium on Principles of Distributed Computing, 1982, ACM, Montreal, Canada, pp. 60-72.

[B] E. BACH, *Realistic analysis of some randomized algorithms*, in Proc. 19th ACM-SIGACT Symposium on Theory of Computing, May 1987, ACM, New York, NY, pp. 453-461.

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

[CW] J. L. CARTER AND M. N. WEGMAN, *Universal classes of hash functions*, J. Comput. System Sci., 18 (1979), pp. 143-154.

[CG] B. CHOR AND O. GOLDREICH, *On the power of two-point sampling*, J. Complexity, 5 (1989), pp. 96-106.

[KR] H. KARLOFF AND P. RAGHAVAN, *Randomized algorithms, and pseudorandom numbers*, in Proc. 20th ACM-SIGACT Symposium on Theory of Computing, May 1988, ACM, Chicago, IL, pp. 310-321.

[KY] D. E. KNUTH AND A. C. YAO, *The complexity of nonuniform random number generation*, in Algorithms and Complexity, J. E. Traub, ed., Academic Press, New York, 1976, pp. 357-428.

[L] T. LANG, *Interconnections between processors and memory modules using the shuffle-exchange network*, IEEE Trans. Comput. 25 (1976), pp. 496-503.

[R] A. G. RANADE, *How to Emulate a Shared Memory*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1987, pp. 185-194.

[Ul] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.

[Up] E. UPFAL, *Efficient Schemes for Parallel Communication*, J. Assoc. Comput. Mach., 31 (1984), pp. 507-517.

[V] L. G. VALIANT, *A scheme for fast parallel communication*, SIAM J. Comput., 11 (1982), pp. 350-361.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 267-280, April 1990

005

A HEURISTIC ALGORITHM FOR SMALL SEPARATORS IN ARBITRARY GRAPHS*

DAVID A. PLAISTED†

Abstract. Some heuristic random polynomial time algorithms for finding good cuts in arbitrary graphs are presented. A cut is good if there are a small number of edges across the cut and if the cut divides the set of vertices somewhat evenly. The algorithms obtain cuts from solutions to randomly chosen network flow problems based on the input graph. Probabilistic bounds for the goodness of the cut obtained in terms of the goodness of an optimal separator are derived. These bounds are valid for all input graphs. There is reason to think that the algorithm will perform better than the bounds indicate.

Key words. graph algorithms, random algorithms, separators, maximum flow, network flow

AMS(MOS) subject classifications. 68Q20, 68Q25

1. Introduction. For many applications it is useful to be able to decompose a graph G into two parts G_1 and G_2 that have few edges between them. For example, in VLSI design, G_1 and G_2 may be parts of a circuit G and then G_1 and G_2 can be laid out in adjacent parts of a rectangle with few interconnections between them. Or, G may represent relationships between concepts in a knowledge base; then (G_1, G_2) represents a way of hierarchically decomposing the knowledge base into relatively independent subparts. This corresponds to the *small separator problem.* There is not a complete uniformity of terminology about separators in graphs. We will define separators as follows. A *cut* in an n vertex graph $G=(V, E)$ is a partition $(W, \bar{W})$ of V. An (*edge*) *separator* in an n vertex graph $G=(V, E)$ is a partition $(W, \bar{W})$ of V such that $|W| \geqq n/3$ and $|\bar{W}| \geqq n/3$, where $|A|$ is the number of elements in a set A. Such a separator is called *small* if the number of edges (v, w) with v in W and w in $\bar{W}$ is small. A *vertex separator* is a set of vertices that disconnect a graph into roughly equal components. There are slight variations in these definitions in the following references. Lipton and Tarjan [1979] show that small vertex separators can be found in linear time for planar graphs. However, their method does not guarantee a separator within some bound of an optimal one. Despite this, their algorithm may be used to solve some problems efficiently on planar graphs using a divide-and-conquer approach (Lipton and Tarjan [1977]). Bui et al. [1984] give a probabilistic algorithm that finds optimal edge separators in arbitrary graphs or else halts without output, and that halts without output with low probability for certain classes of graphs. Their algorithm is based on solving network flow problems and finding min cuts using the max flow-min cut theorem of Ford and Fulkerson [1962]. However, it is not known whether this algorithm performs well on arbitrary graphs, or on randomly chosen graphs. Rao [1987] gives an algorithm for finding nearly optimal edge separators in planar graphs. Also, Rajasekaran and Reif [1988] have shown how a small separator algorithm can be used to speed up simulated annealing in some cases. The problem of finding an optimal edge separator in an arbitrary graph is NP-complete (Garey, Johnson, and Stockmeyer [1976]). For a discussion of flow algorithms on networks see Mehlhorn [1984]. A recent algorithm by Leighton and Rao [1988] finds a separator very close

* Received by the editors February 16, 1988; accepted for publication July 6, 1989. This research was supported in part by the National Science Foundation under grant DCR-8516243.

† Department of Computer Science, University of North Carolina at Chapel Hill, Chapel Hill, North Carolina 27599.

to optimum (within a $\log(n)$ factor). This algorithm is based on multicommodity flow and uses linear programming with n^2e variables, where n is the number of vertices and e is the number of edges in the graph. This can be as many as n^4 variables. The best linear programming algorithms have a time bound of $O(n^{14}L)$ arithmetic operations on $O(L)$ bit numbers to solve such systems (Karmarkar [1984]), where L can be as large as n^2. We present an algorithm that uses single commodity flow. It turns out that our algorithm solves $O(n \log^2(n))$ single commodity flow problems and runs in $O(n^4 \log^2(n))$ time since single commodity maximum flow can be found in time $O(n^3)$ (Mehlhorn [1984]). Thus, although we do not come nearly as close to an optimal separator as the algorithm of Leighton and Rao [1988], the worst-case time bound is much better. From now on we use the term separator to describe an edge separator.

We present probabilistic heuristic algorithms with provably good performance on arbitrary graphs. For an arbitrary graph G with separator having m edges, the algorithms will with high probability, and in polynomial time, find a "near separator" having $f(m)$ edges where f is not too large. More precisely, we have the following results. Both results hold in the limit for large graphs. In principle the algorithms could be modified to make these results hold everywhere, by storing a table of the small graphs and their optimal separators. Let n be the number of vertices in graph G henceforth. Also, if C is a cut $(W, \bar{W})$ of G then $|C|$ is the number of edges in C, that is, the number of edges (v, w) with v in W and w in $\bar{W}$.

DEFINITION. The *balance* $b(C)$ of a cut $C = (W, \bar{W})$ is the ratio $\min(|W|, |\bar{W}|)/n$. Note that if C is a separator then $b(C) \geqq \frac{1}{3}$. A *near separator* is a cut with balance near $\frac{1}{3}$, informally.

THEOREM 0.1. *There is a polynomial time algorithm that, given a graph G with a separator S and given a real number r, $0 < r < 1$, with probability .5 outputs a cut C of G such that either C is a separator with at most $3|S|\sqrt{n}/2(1-r)$ edges or C has balance at least $r/3$ and has at most $|S|\sqrt{n}/(1-r)$ edges.*

Proof. See Theorem 5 and its corollary.

THEOREM 0.2. *There is a polynomial time algorithm that, given a graph G with a separator S and given a real number r, $0 < r < 1$, with probability .5 outputs a cut C of G such that either C is a separator with at most $6|S|^2(\ln(n)+1)/(1-r)$ edges, or C is not a separator, has balance at least $r/3$, and has at most $4|S|^2(\ln(n)+1)/(1-r)$ edges.*

Proof. See Theorem 8 and its corollary. Here and throughout we use ln for natural logarithm.

These algorithms perform independently on successive trials, so the probability .5 can be made arbitrarily close to 1 by running the algorithms repeatedly. There is reason to believe that the algorithms will perform much better in practice than the bounds indicate, since the bounds are based on a kind of probabilistic worst-case analysis. Therefore it would be interesting to implement the algorithms and see how well they actually perform.

We summarize the different algorithms here. Algorithm A solves a parameterized flow problem and finds a cut of G. Algorithm A_{max} applies Algorithm A number of times and picks the "best" output. Algorithm A_{max} has a better bound than Algorithm A. Algorithm A* calls Algorithm A on a sequence of graphs to obtain a sequence of cuts of G that are pieced together to produce a near separator of G. Algorithm B is like Algorithm A, but the flow problems are defined differently to better approximate a normal distribution of certain quantities related to small cuts in G. Algorithm B outputs a cut of G. Algorithm B* calls Algorithm B on a sequence of graphs to obtain a sequence of cuts of G that are pieced together to product a near separator of G. The bounds of Theorems 0.1 and 0.2 above refer to Algorithms A* and B*, respectively.

2. The basic approach: Algorithm A. The algorithms we present, like that of Bui et al. [1984], are based on solving network flow problems and using the max flow-min cut theorem. For some recent polynomial time algorithms for finding a maximum flow, see Mehlhorn [1984]. The algorithms we present are very similar to each other; the difference is mainly in the analysis. We now describe Algorithm A. In this algorithm, a random network flow problem based on G is constructed. This problem p is parameterized, so we have $p(x)$, where x is the parameter. This problem is solved using possibly the network flow algorithm of Ford and Fulkerson [1962] or more recent refinements of it, for various values of x, until a "critical value" x' of x is obtained. The maximum flow for $p(x')$ yields a cut C of G by the max flow-min cut theorem. The "goodness" of this cut C is related to the number of edges in a small (edge) separator in G. We will define "goodness" in § 2.1. Now, C may not be a separator, since it may partition the vertices into two very unevenly sized subsets. However, it is possible to perform algorithm A on the larger subset and repeat this process a number of times. By piecing together these cuts in the right way, a near separator may be obtained if desired, with only a slight degradation of the goodness of the resulting separator. This is the idea of Algorithm A*. For some applications, it may suffice to obtain a good cut and it may not be necessary to use Algorithm A*.

The intuitive idea of Algorithm A is as follows: Given a connected graph G, the flow problem p is constructed by randomly choosing half the vertices of G as sources and half the vertices of G as sinks. Let V_A denote the set of sources and V_Z the set of sinks. Also, we add a new source vertex A with edges from A to all sources, and a new sink vertex Z with edges from all sinks to Z. Therefore, in reality, A is the only source and Z is the only sink. However, we speak of the vertices in V as sources or sinks also . The capacities of all the edges incident with A or Z is one, and the capacities of all the edges of G is x, in both directions. Thus this flow problem is parameterized by x. Initially, x is chosen very large, and a maximum flow from A to Z is found. We choose x large enough so that none of the edges of G is saturated by this flow. (A value of $n/2$ for x suffices.) Instead, the edges from A to vertices in V_A are saturated, and likewise the edges from vertices in V_Z to Z are saturated. This is always possible since the capacities of these edges from A or to Z are fixed at 1. Then x is reduced (say, by binary search) until we find a value x' such that the flow problem $p(x'+\varepsilon)$ has a maximum flow in which none of the edges of G are saturated, but the flow problem $p(x')$ has a maximum flow that saturates some of the edges of G. Also, the flow problem $p(x'-\varepsilon)$ has a maximum flow that fails to saturate some edge (A, y) and some edge (w, Z). For example, if G has only one edge between a and b, and a is a source and b is a sink, then when $x>1$, the edge (A, a) will be saturated and the edge (b, Z) will be saturated in a maximum flow. When $x=1$, the edge from a to b will also be saturated in a maximum flow. Thus, the problem $p(1+\varepsilon)$ has a maximum flow with none of the edges of G saturated, but $p(1)$ has a maximum flow that saturates some of the edges of G. Also, $p(1-\varepsilon)$ has a maximum flow that fails to saturate the edges (A, a) and (b, Z). Since x is initially n, $\log(n)$ iterations will reduce the uncertainty to 1, and each further iteration will add a significant bit of information. Thus $O(\log(n))$ iterations will suffice to get a constant number (or even a logarithmic number) of significant digits in the value of the flow.

Let A' denote the set of vertices y of G such that the edge (A, y) is not saturated in a maximum flow for $p(x'-\varepsilon)$. Let Z' denote the set of vertices y of G such that the edge (y, Z) is not saturated in this flow. We can show that the saturated edges of G disconnect A' from Z', and that A' and Z' are nonempty. Therefore, the saturated edges of G disconnect G, and from them we can obtain a cut C separating A' from

Z' such that all the flow across this cut is in one direction. This cut can be obtained easily as in typical maximum flow algorithms. This completes the description of Algorithm A. It should be clear that this can be done in polynomial time since a maximum flow can be found in polynomial time and so Algorithm A takes polynomial time.

The idea of the analysis is as follows: Consider an arbitrary cut $(W, \bar{W})$ of G. What is the chance that this cut will have all edges saturated, which is a prerequisite for its being chosen by Algorithm A? If the cut has many edges, the chance that they will be saturated is low. However, if the cut has few edges, the chance that they will be saturated is high. More precisely, if W and $\bar{W}$ are nearly equal (as with a separator) then there is a reasonably high probability that the number of sources and sinks in W differ by some constant multiple of $\sqrt{n}$ by properties of the normal (or binomial) distribution. However, the probability that these numbers differ by more than a few multiples of $\sqrt{n}$ is extremely low.

DEFINITION. If the cut C is $(W, \bar{W})$ then let Δ_C (equivalently, $\Delta_{(W,\bar{W})}$) be the difference between the number of sources and sinks in W (or $\bar{W}$) in the flow problem $p(x)$.

DEFINITION. The *capacity per edge* through (or of) a cut C is $\Delta_C/|C|$. (Recall that $|C|$ is the number of edges across C.) The *absolute capacity per edge* of a cut C is the absolute value of the capacity per edge of C. We use $|x|$ to refer to the absolute value of a number x, as usual.

THEOREM 1. *Algorithm* A *will always choose some cut having nearly maximum absolute capacity per edge.*

Proof. Let $\phi(x)$ be a maximum flow for the problem $p(x)$. Algorithm A finds x' such that $p(x'+\varepsilon)$ has a maximum flow $\phi(x'+\varepsilon)$ in which no edges of G are saturated (and all edges incident on A and Z are saturated), but $p(x')$ has a maximum flow $\phi(x')$ in which some edges of G are saturated, and $p(x'-\varepsilon)$ has a maximum flow in which some edges incident on A and Z are not saturated. Recall that A' are the vertices y in V_A such that the edge (A, y) is not saturated in $\phi(x'-\varepsilon)$ and Z' are the vertices y in V_Z such that the edge (y, Z) is not saturated in $\phi(x'-\varepsilon)$. Algorithm A finds a minimal cut $(W, \bar{W})$ separating A' from Z'. Assume the flow goes from W to $\bar{W}$, so that $A' \subset W$ and $Z' \subset \bar{W}$. It follows that all sinks in W are saturated (absorbing a flow of 1) and all sources in $\bar{W}$ are saturated (producing a flow of 1) in $\phi(x'-\varepsilon)$.

The total flow in $\phi(x'+\varepsilon)$ is $n/2$ since no edges of G are saturated. By multiplying all flows in $\phi(x+\varepsilon)$ by $(x'-\varepsilon)/(x'+\varepsilon)$ we obtain a (nonmaximum) flow for the problem $p(x'-\varepsilon)$, since the edge capacities are reduced by a ratio of at most $(x'-\varepsilon)/(x'+\varepsilon)$ in going from $p(x'+\varepsilon)$ to $p(x'-\varepsilon)$. This nonmaximum flow has a total flow of $n(x-\varepsilon)/2(x+\varepsilon)$. Therefore the maximum flow $\phi(x'-\varepsilon)$ has a total value of at least $n(x'-\varepsilon)/2(x'+\varepsilon)$. Thus the total flows for $\phi(x'-\varepsilon)$ and $\phi(x'+\varepsilon)$ are about the same.

Now, the total flow for $\phi(x'-\varepsilon)$ is $f_{11}+f_{12}+f_{22}$, where f_{11} is the flow from sources in W (including A') to sinks in W, f_{12} is the flow from sources in W to sinks in $\bar{W}$ (including Z'), and f_{22} is the flow from sources in $\bar{W}$ to sinks in $\bar{W}$. We can omit flow from sources in $\bar{W}$ to sinks in W, since all the flow across the cut is in one direction. We justify this informally as follows. If we have a flow from a source to two sinks, this flow can be expressed as a sum of a flow from the source to each sink. By extending this argument, we can express the flow in $\phi(x'-\varepsilon)$ as $f_{11}+f_{12}+f_{22}$.

Let f'_{11}, f'_{12}, f'_{21}, and f'_{22} be similar quantities for $\phi(x'+\varepsilon)$. By above remarks, $f_{11}+f_{12}+f_{22}$ is nearly n and $f'_{11}+f'_{12}+f'_{21}+f'_{22}=n$. Note that the total flow from sources in W in $\phi(x'-\varepsilon)$, is about the same as the total flow from sources in W in $\phi(x'+\varepsilon)$.

This is because the sources in $\bar{W}$ produce a flow of 1 each in both $\phi(x'-\varepsilon)$ and $\phi(x'+\varepsilon)$, hence the total flow from sources in $\bar{W}$ is the same in $\phi(x'-\varepsilon)$ and $\phi(x'+\varepsilon)$. Also, the total flow in $\phi(x'-\varepsilon)$ is about the same as in $\phi(x'+\varepsilon)$. This flow must come from sources in W and sources in $\bar{W}$. Thus $f_{11}+f_{12}$ is about the same as $f'_{11}+f'_{12}$. Similarly, $f_{12}+f_{22}$ is about the same as $f'_{12}+f'_{22}$. We obtain that f_{12} is about $f'_{12}-f'_{21}$ by combining these two approximate equalities with the fact that the total flows in $\phi(x'-\varepsilon)$ and $\phi(x'+\varepsilon)$ are about the same. Thus the flow through $(W, \bar{W})$ in $\phi(x'-\varepsilon)$ and $\phi(x'+\varepsilon)$ is about the same, since the flow across $(W, \bar{W})$ in $\phi(x'-\varepsilon)$ is f_{12} and in $\phi(x'+\varepsilon)$ it is $f'_{12}-f'_{21}$. But the flow through $(W, \bar{W})$ in $\phi(x'+\varepsilon)$ is $\Delta_{(W,\bar{W})}$, so the flow through $(W, \bar{W})$ in $\phi(x'-\varepsilon)$ is about $\Delta_{(W,\bar{W})}$. Since each edge of $(W, \bar{W})$ is saturated in $\phi(x'-\varepsilon)$, the flow through each edge of $(W, \bar{W})$ in $\phi(x'-\varepsilon)$ is $x'-\varepsilon$. Therefore $(x'-\varepsilon)|(W, \bar{W})|$ is nearly $\Delta_{(W,\bar{W})}$ and the capacity per edge of $(W, \bar{W})$ is nearly $x'-\varepsilon$. Since Algorithm A chooses X' nearly maximum so that some such cut $(W, \bar{W})$ exists, Algorithm A finds a cut in G of nearly maximum capacity per edge.

The idea, then, is that if a cut $(W, \bar{W})$ has a few edges, then in a random selection of sources and sinks, its expected capacity per edge will be high, while if a cut has many edges, its expected capacity per edge will be low. Since Algorithm A always chooses a cut having nearly maximal capacity per edge, it should always choose a cut having nearly the smallest number of edges. However, this analysis leaves out some factors. Namely, if W or $\bar{W}$ is very small then the expected value of $\Delta_{(W,\bar{W})}$ is also small, and so even if the cut $(W, \bar{W})$ has a small number of edges, it may have a small expected capacity per edge. Therefore the measure of goodness must also take into account the relative sizes of W and $\bar{W}$. Also, even if the probability is low that any particular cut has high capacity per edge, if there are many cuts, then the collective probability that one of them has a high capacity per edge, may be high. Therefore we need to do an analysis of the number of cuts of various sizes in a connected graph. This turns out to be fairly easy.

2.1. A bound on the performance of Algorithm A. We first give a fairly simple analysis to produce a rough bound on the performance of Algorithm A. Let $C(n, m)$ be the binomial coefficient giving the number of ways of choosing m out of n objects. Thus $C(n, m) = n!/m!(n-m)!$. Our analysis makes use of the following fact.

THEOREM 2. *Suppose half of the vertices of G are chosen randomly as sources and the other half as sinks. Suppose $C = (W, \bar{W})$ is a cut in G and $|W| = n_1$ and $|\bar{W}| = n_2$. Thus $n_1 + n_2 = n$. Then as n approaches infinity with n_1/n held to a constant with $0 < n_1/n < 1$, the distribution of Δ_C approaches a normal distribution with mean 0 and standard deviation $\sqrt{n_1 n_2/n}$.*

Proof. The number of ways of choosing sources and sinks so that Δ_C is $2d$, is $C(n_1, (n_1/2)+d)C(n_2, (n_2/2)-d)$. The total number of ways of choosing sources and sinks is $C(n, n/2)$. Therefore the probability that $\Delta_C/2 = d$ is

$$\frac{C(n_1, n_1/2+d)C(n_2, n_2/2-d)}{C(n, n/2)}.$$

Now, the binomial distribution $C(n, m)$ may be approximated by a normal distribution. We have that

$$\frac{C(n, m)}{2^n} \to \sqrt{\frac{2}{\pi n}}\, e^{-2(m-n/2)^2/n}$$

as $n \to \infty$ with $(m-n/2)^2/n$ held constant. This is a normal distribution of random

variable m, with mean $n/2$ and standard deviation $2\sqrt{n}$. It follows that the distribution of $\Delta_C/2$ approaches a normal distribution with mean 0 and standard deviation $(\frac{1}{2})\sqrt{n_1 n_2/n}$ as n_1 and n_2 approach infinity with n_1/n held constant. Therefore the distribution of Δ_C is a normal distribution with mean 0 and standard deviation $\sqrt{n_1 n_2/n}$.

COROLLARY 1. *With C as above, the distribution of Δ_C/n_1 approaches a normal distribution with mean 0 and standard deviation $\sqrt{(1/n_1)-(1/n)}$ as n approaches infinity with n_1/n held constant. Also, this standard deviation decreases as n_1 increases.*

Proof. The standard deviation of Δ_C/n_1 is $(1/n_1)\sqrt{n_1 n_2/n}$, which equals $\sqrt{(1/n_1)-(1/n)}$.

COROLLARY 2. *With C as above, if C is a separator in G then the probability that $|\Delta_C| \geqq (\frac{1}{3})\sqrt{2n}$ approaches a constant greater than .3 as n approaches infinity. Also, regardless of whether C is a separator in G, if $n_1 \leqq n_2$ then the probability that $|\Delta_C|/n_1 \geqq 1/\sqrt{n}$ approaches a constant greater than .3 as n approaches infinity.*

Proof. Note that since n_1 and n_2 are at least $n/3$, and $n_1+n_2=n$, the standard deviation is at least $(\frac{1}{3})\sqrt{2n}$. The probability that Δ_C has absolute value larger than the standard deviation is at least .3, from tables for the normal distribution. Now, the ratio of the standard deviation to n_1 is minimal (assuming $n_1 \leqq n_2$) when $n_1 = n_2$ and then it has a value $1/\sqrt{n}$. Therefore the probability that $|\Delta_C|/n_1 \geqq 1/\sqrt{n}$ is at least .3 for large n.

From now on we assume that $n_1 \leqq n_2$.

DEFINITION. The *goodness* $g(C)$ of a cut $C=(W, \bar{W})$ is the ratio $\min(|W|, |\bar{W}|)/|C|$.

This is equivalent to flux or minimum edge expansion used in VLSI. Also, a separator of maximum goodness is similar to a minimum quotient separator discussed by Miller [1984] and Rao [1987].

DEFINITION. If $C=(W, \bar{W})$ is a cut, then W and $\bar{W}$ are called *sides* of C.

THEOREM 3. *Suppose G has a cut $C=(W, \bar{W})$. Thus as n, $|W|$, and $|\bar{W}|$ approach infinity, the probability that Algorithm* A *will find a cut having goodness at least about $g(C)/\sqrt{n}$ approaches a constant greater than .3. Also, this probability increases as the balance of C decreases.*

Proof. In Corollary 2 above we have shown that the probability that $|\Delta_C|/n_1 \geqq 1/\sqrt{n}$ is at least .3 for large n. The algorithm will always find a cut having nearly maximum absolute capacity per edge. The capacity per edge of C is $\Delta_C/|C|$. Therefore Algorithm A finds some cut D whose absolute capacity per edge is at least about $|\Delta_C|/|C|$. Thus the total flow through D is at least about $|D||\Delta_C|/|C|$. This means that there are at least about $|D||\Delta_C|/|C|$ vertices on either side of side of D. Hence $g(D)$ is at least about $|\Delta_C|/|C|$. But $g(C)$ is $n_1/|C|$. Thus $g(D)/g(C)$ is at least about $|\Delta_C|/n_1$. However, the probability that $|\Delta_C|/n_1 \geqq \sqrt{n}$ is at least .3 for large n, as shown above. Therefore the probability that $g(D)/g(C)$ is at least about $1/\sqrt{n}$ is at least .3, for large n. Thus the probability that $g(D)$ is at least about $g(C)/\sqrt{n}$ is at least .3 for large n, as claimed. Since cuts C with small balance have larger standard deviation of Δ_C, by Corollary 1 above, the probability increases as the balance of C decreases. It is necessary that $|W|$ and $|\bar{W}|$ approach infinity so that we can approximate the distribution of Δ_C by a normal distribution. Note that C need not be a separator.

3. A better bound: Algorithm $\mathbf{A_{max}}$. The probability that $|\Delta_C|/n_1 \geqq 1/\sqrt{n}$ is at least .3 for large n. However, sometimes $|\Delta_C|/n_1$ will be much larger than this, since this quantity has a normal distribution. For example, from tables for the normal distribution, the probability that a random normally distributed variable with mean zero has absolute values at least twice the standard deviation, is at least .045, and the probability that

such a random variable has absolute value at least three times the standard deviation is at least .0025. Therefore, in 25 trials we will probably obtain at least one result having absolute value at least twice the standard deviation, and in 400 trials we will probably obtain at least one absolute value at least three times the standard deviation. Thus, if Algorithm A is run say 25 times, and the best cut is kept, we will probably obtain a cut D having goodness at least about $2g(C)/\sqrt{n}$, and in 400 trials we will probably obtain a cut D having goodness at least about $3g(C)/\sqrt{n}$. In this way, at the expense of running more trials, we can get a better bound than that of Algorithm A. Reasoning in this way, we obtain the following result.

THEOREM 4. *Suppose G has a cut C. Then as $n \to \infty$, if Algorithm* A *is run $\Theta(k\, e^{k^2/2})$ times and the best cut D is taken, the probability is greater than .5 that $g(D)/g(C)$ is at least about $k/\sqrt{n}$.*

Proof. We can show that $\int_{x_0}^{\infty} e^{-t^2/2}\, dt$ is $\Theta((1/x_0)\, e^{-x_0^2/2})$. Therefore the probability that a normally distributed variable with mean zero, has absolute value at least k times the standard deviation, is $\Theta((1/k)\, e^{-k^2/2})$. Therefore the probability that a cut found by the algorithm has goodness at least about $kg(C)/\sqrt{n}$ is $\Theta((1/k)\, e^{-k^2/2})$. Therefore in $\Theta(k\, e^{k^2/2})$ trials, the probability is greater than .5 that the best cut will have goodness at least about $kg(C)/\sqrt{n}$.

We will derive a much better bound on $g(D)$ below for separators C such that $g(C) < \sqrt{n}$. First we consider how the cuts found by the algorithm can be pieced together to form a "near separator."

4. Piecing together cuts: Algorithm A*. Suppose $C = (W, \bar{W})$ is an arbitrary cut in G. We present an algorithm for constructing a "near separator" in G and bound the goodness of the near separator obtained in terms of the goodness of C. This method is the same as that used in Rao [1987] for planar graphs. Algorithm A* constructs a sequence $G_0, G_1, G_2, \cdots$ of graphs. G_0 is G. G_i is (V_i, E_i). G_{i+1} is obtained from G_i by finding a good cut (Y_i, Z_i) in G_i using repeated calls to Algorithm A. (We could, of course, also use Algorithm A_{max}, obtaining an Algorithm A^*_{max}.) Assume without loss of generality that $|Y_i| \leqq |Z_i|$. Then V_{i+1} is $V_i - Y_i$ and E_{i+1} is all edges of E_i with both endpoints in V_{i+1}. From this sequence of graphs we define a sequence $(W_1, \bar{W}_1)$, $(W_2, \bar{W}_2)$, $(W_3, \bar{W}_3) \cdots$ of cuts of G. The output of Algorithm A* is one of the cuts $(W_i, \bar{W}_i)$ that may be chosen from the sequence in various ways. We analyze the goodness of this output cut in terms of the method of choosing it from the sequence. The sequence of cuts is defined as follows: W_1 is Y_1 and, in general, W_{i+1} is $W_i \cup Y_{i+1}$. Thus W_i is a union of the smaller sides of the cuts (Y_i, Z_i). Note that the length of the sequence G_i of graphs is at most $n+1$ since V_{i+1} is smaller than V_i always. We stop the sequence when $|W_i| \geqq n/3$ and so the length of the sequence is at most $(n/3)+1$.

We must specify how the "good cut" (Y_i, Z_i) is found. In general, Algorithm A is used. However, it is necessary to apply this algorithm a number of times and take the best cut, to insure that the probability of finding a sufficiently good cut is sufficiently high. By Theorem 3 above, if Algorithm A is applied once it obtains a cut within a ratio of $1/\sqrt{n}$ to optimal, with probability greater than .3. Therefore if Algorithm A is applied twice, and the best result taken, we obtain a cut within a ratio of $1/\sqrt{n}$ to optimal, with probability greater than .5. If Algorithm A is applied $2\log_2(n/3)$ times and the best cut taken, we obtain a cut within a ratio of $1/\sqrt{n}$ to optimal with probability greater than $1-1/(n/3)$. Algorithm A* depends on all the cuts (Y_i, Z_i) being within a ratio of $1/\sqrt{n}$ to an optimal cut in G_i. The probability of this is at least $(1-1/(n/3))^{n/3}$ since the sequence has length at most $n/3$. However, $(1-1/(n/3))^{n/3} > .5$. Therefore the probability that all the cuts (Y_i, Z_i) will be within a ratio of $\sqrt{n}$ to optimal in their

graphs, is greater than .5. Note that the total number of trials for this is at most $(2n/3)\log_2(n/3)$. Since each application of Algorithm A may require $O(\log(n))$ flow problems to be solved, and a single flow problem takes $O(n^3)$ time, the total time for Algorithm A* is $O(n^4\log^2(n))$.

We also need to analyze how many edges the cuts (Y_i, Z_i) will have. We know that they will, with high probability, have goodness within a ratio of $1/\sqrt{n}$ to an optimal cut in G_i. However, the optimal cut in G_i may not be as good as the optimal cut in G. Let m be $|(W, \bar{W})|$ and let n_i be $|V_i|$. Suppose that $|W_{i-1}| < |W|$. Then the graph G_i has a cut $(W \cap V_i, \bar{W} \cap V_i)$ with at most m edges and at least $|W| - |W_{i-1}|$ vertices in each part. The goodness of this cut is at least $(|W| - |W_{i-1}|)/m$. Therefore with high probability, by Theorem 3, $g((Y_i, Z_i)) \geqq (1/\sqrt{n_i})(|W| - |W_{i-1}|)/m$, assuming $|W_{i-1}| < |W|$. Since $n_i \leqq n$, $g((Y_i, Z_i)) \geqq (1/\sqrt{n})(|W| - |W_{i-1}|)/m$ with high probability. Since $|Y_i| \leqq |Z_i|$, $|(Y_i, Z_i)|$ is at most $|Y_i|/g((Y_i, Z_i))$, that is, at most $|Y_i|m\sqrt{n}/(|W| - |W_{i-1}|)$.

We now analyze how good the cuts $(W_i, \bar{W}_i)$ will be. The number of edges in the cut $(W_i, \bar{W}_i)$ of G is at most the sum of the number of edges in the cuts (Y_1, Z_1), $(Y_2, Z_2), \cdots, (Y_i, Z_i)$. Thus $|(W_i, \bar{W}_i)| \leqq m\sqrt{n}\sum_{j=1}^{i} |Y_j|/(|W| - |W_{j-1}|)$. Suppose for this discussion that $|W_i| \leqq |W|$. Since $|W_{j-1}|$ is the sum of $|Y_k|$ for k from 1 to $j-1$, we can show that $|(W_i, \bar{W}_i)| \leqq m\sqrt{n}(H_{|W|} - H_{|W|-|W_i|})$ where H_p is $1+\frac{1}{2}+\frac{1}{3}+\cdots+1/p$. Therefore $|(W_i, \bar{W}_i)| \leqq m\sqrt{n}H_{|W|}$. Now H_p is $\ln(p)+\gamma+O(1/p)$ where $\gamma = .5772\cdots$ is Euler's constant. We obtain therefore (assuming $|W_i| \leqq |W|$) that $|(W_i, \bar{W}_i)| \leqq m\sqrt{n}(\ln(|W|)+\gamma+O(1/|W_i|))$. Let g_i be $g((W_i, \bar{W}_i))$. Now, if $|W_i| \leqq |W|$ then $|W_i| \leqq |\bar{W}_i|$ so $g_i = |W_i|/|(W_i, \bar{W}_i)|$. Since $g((W, \bar{W})) = |W|/m$, $g_i \geqq (|W_i|/|W|)g((W, \bar{W}))/(\sqrt{n}\ln(|W|+\gamma+O(1/|W_i|))$. For another bound, recall from above that $|(W_i, \bar{W}_i)| \leqq m\sqrt{n}(H_{|W|} - H_{(|W|-|W_i|)})$. We can show that if $p < q$, then $H_q - H_{(q-p)} \leqq p/(q-p)$. If $p \leqq q$ then $1/(H_q - H_{(q-p)}) \geqq (q/p) - 1$. Therefore, using the inequality $|(W_i, \bar{W}_i)| \leqq m\sqrt{n}(H_{|W|} - H_{|W|-|W_i|})$ from above, $1/(W_i, \bar{W}_i)| \geqq (1/(m\sqrt{n}))(|W|/|W_i| - 1)$. With g_i as above, $g_i \geqq (|W_i|/(m\sqrt{n}))(|W|/|W_i| - 1)$, and $g_i \geqq (|W_i|/|W|)g((W, \bar{W}))(|W|/|W_i| - 1)/\sqrt{n}$ so $g_i \geqq g((W, \bar{W}))(1 - |W_i|/|W|)/\sqrt{n}$.

We now bound the goodness of the cut produced (with probability $>.5$) by Algorithm A*. We shown above that if Algorithm A* stops at graph G_i such that $|W_i| \leqq |W|$, then with probability greater than .5 it outputs a cut $(W_i, \bar{W}_i)$ with goodness at least $(|W_i|/|W|)g((W, \bar{W}))/(\sqrt{n}\ln(|W|)+\gamma+O(1/|W_i|))$. Also, by the second bound, $g(W_i, \bar{W}_i) \geqq g((W, \bar{W}))(1 - |W_i|/|W|)/\sqrt{n}$. The first estimate is good when $|W_i|$ is large, the second when $|W_i|$ is small. These may be combined by using the first estimate when $|W_i| \geqq |W|/2$ and the second when $|W_i| \leqq |W|/2$. In this way, assuming that $|W| \geqq 2$ we obtain that $g(W_i, \bar{W}_i) \geqq g(W, \bar{W})/(2\sqrt{n}(\ln|W|+\gamma+O(1/|W_i|)))$ with probability greater than .5, if $|W_i| \leqq |W|$. Thus we have an extra cost of about $\ln|W|$ from piecing together the cuts. The problem with this bound is that $|W_i|$ may be very small and so the cut $(W_i, \bar{W}_i)$ may not be a near separator.

To guarantee that a cut is found in which $|W_i|$ is not too small, we can take the first cut $(W_i, \bar{W}_i)$ such that $|W_i|/|W| > r$ for some real number r less than one. In this case we may have $|W_i| > |W|$. We can show that with probability greater than .5 the cut $(W_i, \bar{W}_i)$ produced has goodness at least $(1-r)g(W, \bar{W})/\sqrt{n}$. Thus the cost increases as the ratio r approaches 1. To obtain this bound, recall that $g_i \geqq g((W, \bar{W}))(1 - |W_i|/|W|)/\sqrt{n}$ if $|W_i| \leqq |W|$. (All bounds are with appropriate probabilities.) Also, recall that $g((Y_i, Z_i)) \geqq (1/\sqrt{n})(|W| - |W_{i-1}|)/m$ if $|W_{i-1}| < |W|$. Suppose that $|W_{i-1}| \leqq r|W|$ and $|W_i| > r|W|$. Then $|W_{i-1}| < |W|$, so by the bound on $g((Y_i, Z_i))$, $g((Y_i, Z_i)) \geqq g(W, \bar{W})(1/\sqrt{n})(1 - |W_{i-1}|/|W_i|)$. Also, $g_{i-1} \geqq g((W, \bar{W}))(1 - |W_{i-1}|/|W|)/\sqrt{n}$. These bounds are identical. Since g_i is a weighted average of $g((Y_i, Z_i))$ and g_{i-1}, it follows that $g_i \geqq g((W, \bar{W}))(1 - |W_{i-1}|/|W|)/\sqrt{n}$, that

is, $g_i \geqq (1-r)g(W, \bar{W})/\sqrt{n}$. Also, since $|W_i| > r|W|$, we know that $|W_i|$ is not extremely small.

A problem is that we may not know $|W|$ and so we may not know when to stop. However, if $(W, \bar{W})$ is a separator then we know that $|W| \geqq n/3$ and so we can output the first cut $(W_i, \bar{W}_i)$ such that $|W_i| \geqq r(n/3)$. We can show that the above bound on g_i still holds in this case. Also, $|\bar{W}_i|$ is at least $n/3$ since W_i is $W_{i-1} \cup Y_i$, $|W_{i-1}| < rn/3$, $W_{i-1} \cup Y_i \cup Z_i$ is V, $|Y_i| \leqq |Z_i|$, and $\bar{W}_i$ is Z_i. If $|W_i| \leqq |Z_i|$ then the balance is at least $r/3$, otherwise it is at least $\frac{1}{3}$. Thus, since $r < 1$, the balance of $(W_i, \bar{W}_i)$ is not less than $r/3$ and $(W_i, \bar{W}_i)$ is a near separator. Therefore we have the following result.

THEOREM 5. *Suppose graph G has a separator $(W, \bar{W})$. Suppose Algorithm* A* *is used on G, real number r with $0 < r < l$ is given, and i is minimal such that $|W_i| \geqq r(n/3)$. Then the balance of $(W_i, \bar{W}_i)$ is at least $r/3$, and $g((W_i, \bar{W}_i)) \geqq (1-r)g(W, \bar{W})/\sqrt{n}$ with probability greater than .5.*

Proof. The proof is given above.

COROLLARY. *If G has a separator C, Algorithm* A* *is used, real number r with $0 < r < 1$ is given, and i is minimal such that $|W_i| \geqq r(n/3)$ for some real number $r < 1$, then either $(W_i, \bar{W}_i)$ is a separator with at most $3|C|\sqrt{n}/2(1-r)$ edges or $(W_i, \bar{W}_i)$ has balance at least $r/3$ and has at most $|C|\sqrt{n}/(1-r)$ edges, with probability greater than .5. Thus we obtain a near separator with $O(|C|\sqrt{n})$ edges.*

Proof. Using the definition of goodness for the separator C and the cut $(W_i, \bar{W}_i)$ and the fact that if $(W_i, \bar{W}_i)$ is not a separator then $\min(|W_i|, |\bar{W}_i|)/\min(|W|, |\bar{W}|) \leqq 1$. If $(W_i, \bar{W}_i)$ is a separator then this quantity is bounded by $\frac{3}{2}$.

5. A more refined analysis: Algorithm B. We now give a more refined analysis on a slightly different algorithm to produce a better bound. Note that if $\Delta_C/n_1 \geqq 1/\sqrt{n}$ then Algorithm A will always find a cut D as specified in Theorem 3. However, we can get a better bound by looking at the goodness of cuts D that are often found, rather than always found, when $\Delta_C/n_1 \geqq 1/\sqrt{n}$. For, the probability that a cut D as above will have capacity per edge $\Delta_C/|C|$ will be very small if D has many more edges than C. Thus if there are not too many cuts with many more edges than C, we will probably find one having not too many edges. We makes this precise below. To do this we need to consider the number of cuts of G of various sizes. Then, after showing that the probability is low that a cut with many edges will be chosen, we still must show that among the cuts with few edges, the probability that a cut of low goodness will be chosen is low. It turns out that this is also possible.

For this algorithm, we need to change the flow so that even for C with small balance, the distribution of Δ_C approximates a normal distribution. Suppose $C = (W, \bar{W})$ and $|W| = 1$. Then either $\Delta_C = 1$ or $\Delta_C = -1$ and this does not approximate a normal distribution. To overcome this, we choose a number K and let each node of G contain K *subnodes.* Half of the subnodes in G are chosen as sources and half are chosen as sinks. So, with C as above with one side containing only one vertex, Δ_C will be an integer in the range $-K$ to K. If K is large enough, even for such a C, Δ_C will approximate a normal distribution. In actuality, there are no subnodes, but the flow out of C is some integer in the range $-K$ to K. Other than this, the algorithm is as before. That is, Algorithm B is like Algorithm A with this change. The graph G is also as before; the edges of G connect nodes of G, not subnodes. Only the choice of flow out of nodes of G differs.

Suppose that a cut S, not necessarily a separator, exists in G. Suppose that the balance $b(S)$ of S is b. Note that $b \leqq \frac{1}{2}$. Then Δ_s is approximately distributed as a normal distribution with mean 0 and standard deviation $\sqrt{Kb(1-b)n}$. This is by

Theorem 2, using the fact that the number of subnodes on each side of S is Kbn and $Kn - Kbn$, respectively, and $(Kbn)(Kn - Kbn)/Kn = Kb(1-b)n$. Thus the probability that $|\Delta_S| \geqq \sqrt{Kb(1-b)n}$ is larger than .3 for large K and n.

Suppose S has p edges. Then the absolute capacity per edge of S is $\geqq (1/p)\sqrt{Kb(1-b)n}$ with probability greater than .3. For another cut C to be chosen when $|\Delta_S| \geqq \sqrt{Kb(1-b)n}$, the absolute capacity per edge of C must be at least about $(1/p)\sqrt{Kb(1-b)n}$. Suppose that C has m edges. For the absolute capacity per edge of C to be this large, $|\Delta_C|$ must be at least about $(m/p)\sqrt{Kb(1-b)n}$. Suppose the sides of C have n_1 and n_2 vertices, respectively, with $n_1 \leqq n_2$. Then the distribution of Δ_C is a normal distribution with mean 0 and standard deviation $\sqrt{Kn_1n_2/n}$ for large n and K. Integrating the tails of the normal distribution, it turns out that the probability that $|\Delta_C|$ is at least $(m/p)\sqrt{Kb(1-b)n}$, that is, the probability that cut cut C with m edges might be chosen instead of S when Δ_S is as above, is at most

$$(p/m)(1/\sqrt{2\pi b(1-b)})\, e^{-2b(1-b)m^2/p^2}.$$

Let us call this quantity $P_0(m)$. We must consider the number of cuts C with m edges to find a bound on the probability that one of them with small goodness will be chosen instead of S.

5.1. The number of cuts of various sizes. Since G is a connected graph, it has a spanning tree. Let T be a spanning tree of G. Now, there is a one-to-one correspondence between subsets of the edges of T, and cuts of G. Given a cut C of G, from it we can obtain the set $T(C)$ of edges e of T such that the endpoints of e are on different sides of C. Also, given such a set $T(C)$ of edges, we can from it obtain C by starting at some point in T and assigning vertices to one or the other side of C so that the condition on $T(C)$ is satified. (We are identifying the cuts $(W, \bar{W})$ and $(\bar{W}, W)$.) Now, T has $n-1$ edges. Note that $|C| \geqq |T(C)|$. So if $|C| \leqq m$ then $|T(C)| \leqq m$, and we can bound the number of cuts with m edges or less, by bounding the number of subsets of the edges of T, with m elements or less. But the number of subsets of the edges of T with m elements or less, is just $C(n-1,1)+C(n-1,2)+\cdots+C(n-1,m)$ where $C(i,j)$ is a binomial coefficient, as before. Thus the number of cuts of G with m edges or less, is at most $C(n-1)+C(n-1,2)+\cdots+C(n-1,m)$. Looked at another way, let T_m be the set of cuts C such that $|T(C)| = m$. Then all cuts C in T_m satisfy $|C| \geqq m$. Thus we know that the set of cuts of G can be expressed as the union of such T_m where $|T_m| = C(n-1, m)$. In this way we obtain the following result.

THEOREM 6. *The set of cuts of G can be expressed as the union over m of sets T_m of cuts such that* (a) *each cut C in T_m has at least m edges and* (b) $|T_m| = C(n-1, m)$.

Proof. The proof is as above.

5.2. Bounding the probability of choosing a large cut. We first show that the probability is small that a cut will be chosen from some T_m with m large relative to p. We say m is large relative to p if a cut with m edges cannot possibly have a goodness near that of the best cut even if the cut has $n/2$ vertices on each side. Therefore there are a small number of cuts (relative to p) that can be chosen. We next show that of these remaining cuts, the probability is small that any one of them with a small goodness relative to that of S, will be chosen. Therefore there is a large probability that a cut with a goodness near that of S will be chosen.

Now, $|T_m| = C(n-1, m) < (n-1)^m$, which is about $e^{m\ln(n)}$. An arbitrary cut with m edges will be chosen with probability bounded by $P_0(m)$, which is

$$(p/m)(1/\sqrt{2\pi b(1-b)})\, e^{-2b(1-b)m^2/p^2},$$

when $|\Delta_S|$ is large, as specified above. Therefore, since there are about $e^{m\ln(n)}$ such cuts, the probability that any cut with m edges will be chosen when $|\Delta_S|$ is large, is bounded by

$$P_0(m)\, e^{m\ln(n)}.$$

Let $m_0 > 0$ be some fixed value for m; then if $m \geqq m_0$ the probability is bounded by

$$(p/m_0)(1/\sqrt{2\pi b(1-b)})\, e^{-2b(1-b)m^2/p^2+m\ln(n)}.$$

Integrating from m_0 to infinity with respect to m and bounding the result, we show that the probability that any cut in T_m will be chosen for any $m \geqq m_0$ is less than

$$\frac{p^3\sqrt{2}}{m_0\sqrt{\pi b(1-b)}(8m_0b(1-b)-2p^2\ln(n))}\, e^{-2b(1-b)m^2/p^2+m\ln(n)}.$$

Let us call this quantity $P_1(n)$. Assuming that $p^2\ln(n)/(2b(1-b)) < m_0$, we obtain that $P_1(n)$ is bounded by

$$\frac{\sqrt{2b(1-b)}}{\sqrt{\pi}p\ln(n)^2}.$$

Since $\sqrt{\pi} > 1.5$, if $n > 3$ this probability is less than $\frac{1}{2}$, and becomes much smaller than $\frac{1}{2}$ for larger n and $p > 1$. Thus for problems of practical interest this probability is small.

5.3. Bounding the probability of choosing a small cut. We now discuss the cuts in T_m for $m \leqq m_0$. For these cuts, we distinguish those having high goodness from those having low goodness, and show that the probability is low that a cut with low goodness will be chosen. If a C has m edges and goodness g then its sides have gm and $n-gm$ vertices, respectively. Therefore the standard deviation of Δ_C is $\sqrt{Kgm(n-gm)/n}$, and Δ_C is distributed as

$$\sqrt{n/(Kgm(n-gm)2\pi)}\exp\left(\frac{-x^2n}{2Kgm(n-gm)}\right).$$

The cut S will be chosen often unless some other cut has a larger or equal absolute capacity per edge. The absolute capacity per edge of S is at least $\sqrt{Kb(1-b)n}/p$ with probability at least .3, since S has p edges. For the absolute capacity per edge of C to be this large, $|\Delta_C|$ must be at least $(m/p)\sqrt{Kb(1-b)n}$. Therefore the probability that this will occur can be obtained by integrating the above distribution from $(m/p)\sqrt{Kb(1-b)n}$ to infinity and multiplying by two. This yields a value bounded by

$$\frac{p\sqrt{2gm(n-gm)}}{nm\sqrt{\pi b(1-b)}}\exp(-mn^2b(1-b)/(2p^2g(n-gm))).$$

Now, suppose for simplicity that all cuts in T_m have m edges. (If cuts have more edges than this, our algorithm will perform better.) There are less than n^m or $e^{m\ln(n)}$ cuts in T_m. The probability that one of them will have the absolute value of Δ at least $(m/p)\sqrt{Kb(1-b)n}$ is therefore at most $e^{m\ln(n)}$ times the preceding probability. This probability is therefore bounded by

$$\frac{p\sqrt{2gm(n-gm)}}{nm\sqrt{\pi b(1-b)}}\exp((m\ln(n)-mn^2b(1-b))/(2p^2g(n-gm))).$$

Let us call this quantity $P_2(m, n)$. We want to choose g small enough so that the sum

of this quantity from $m=1$ to $m=p^2\ln(n)/(2b(1-b))$, is small. If we choose g such that $g(n-gm)<n^2b(1-b)/(2p^2\ln(n))$, then the exponent is less than zero and $P_2(m,n)$ is bounded by $1/\sqrt{m\pi\ln(n)}$. If we choose g such that $g(n-gm)<n^2b(1-b)/(2p^2(\ln(n)+1))$, then $P_2(m,n)$ is bounded by $e^{-m}/\sqrt{m\pi\ln(n)}$. Summing $P_2(m,n)$ from $m=1$ to $m=\infty$, using the fact that $P_2(m,n)$ is bounded by $e^{-m}/\sqrt{\pi\ln(n)}$, we obtain a total bounded by $1/\sqrt{\pi\ln(n)}$. This quantity becomes small as n increases.

5.4. Bounding the performance of Algorithm B. For large enough n, both this quantity and $P_1(n)$ are small, under the assumption that the absolute capacity per edge of S is at least $(1/p)\sqrt{Kb(1-b)n}$. The absolute capacity per edge of S will be this large with probability at least .3. Therefore we have the following result.

THEOREM 7. *Suppose graph G has a cut S, not necessarily a separator, and Algorithm* B *is applied to G. For large n, there is a probability at least .3 that either the cut S will be chosen by Algorithm* B *or some cut C satisfying $g(C)(n-g(C)|C|)\geqq n^2b(1-b)/(2|S|^2(\ln(n)+1))$ will be chosen, where b is the balance of S.*

Proof. The probability that a large cut will be chosen is $P_1(n)$, which approaches zero as n approaches infinity if m_0 is suitably chosen. The probability that a small cut will be chosen with low goodness is obtained by summing $P_2(m,n)$ with m varying from 1 to infinity. This quantity also approaches zero as n approaches infinity. The absolute capacity per edge of S will be as large as stated, with probability at least 3. Therefore, the algorithm will choose a small cut with a high goodness with probability at least .3, for large n.

Noting that $\frac{1}{2}\leqq(n-gm)/n\leqq 1$ and that bn/p is the goodness of S, $g\geqq g(S)/4p(\ln(n)+1)$. Therefore the goodness of the cut obtained is bounded in terms of the goodness of an optimal cut. The extra factor of p is essentially the cost we pay because there are so many cuts in T_m for small m that it is necessary to choose a small goodness to insure that the total probability of choosing any of them is small. This extra factor will be reduced if many of the cuts in T_m have more than m edges, because the capacity per edge is inversely proportional to the number of edges. For many graphs the cuts in T_m will have m^α edges for some real $\alpha>1$, leading to an even better bound. Also, the behavior of different cuts is highly correlated, with we have not taken into account. This will also improve the bound. We will return to this point shortly. Therefore Algorithm B may do much better in practice than the bound indicates, so implementation would be useful. In addition, the performance of Algorithm B is influenced as much (and even more) by the goodness of cuts that are not separators, as by the goodness of separators. Therefore, if the graph G has any good cut, separator or not, then the algorithm will tend to find a good cut. This also will tend to improve the performance of the algorithm. This completes our description of the basic algorithm.

6. Piecing together cuts to get a separator: Algorithm B*. As before, we can apply Algorithm B a number of times to obtain a near-separator with goodness near that of S. We thus obtain Algorithm B*, which is the same as Algorithm A* except that Algorithm B is used instead of Algorithm A. We then obtain the following result.

THEOREM 8. *Suppose Algorithm* B* *is applied to graph G having a cut $S=(W,\bar{W})$. Suppose also that real number r, $0<r<1$, is given, and i is minimal such that $|W_i|\geqq r(n/3)$. Then the balance of the cut $(W_i,\bar{W}_i)$ is at least $r/3$ and $g((W_i,\bar{W}_i))\geqq(1-r)g(S)/4|S|(\ln n+1)$ with probability greater than .5.*

Proof. Similar to the proof of Theorem 5. Note that the worst case is when the cuts $(W\cap V_i,\ \bar{W}\cap V_i)$ all have $|S|$ edges. This makes the ratio corresponding to $1/(4|S|(\ln(n)+1))$ in Algorithm B a constant and so the previous proof method can be applied.

COROLLARY. *With i as above, and assuming that S is a separator, the cut* $(W_i, \bar{W}_i)$ *has at most* $6|S|^2(\ln(n)+1)/(1-r)$ *edges, and if the cut* $(W_i, \bar{W}_i)$ *is not a separator, it has at most* $4|S|^2(\ln(n)+1)/(1-r)$ *edges, with probability greater than .5.*

Proof. Using Theorem 8, the definition of goodness, and the fact that if $(W_i, \bar{W}_i)$ is not a separator then $\min(|W_i|, |\bar{W}_i|)/\min(|W|, |\bar{W}|) \leqq 1$. If $(W_i, \bar{W}_i)$ is a separator then this quantity is bounded by $\frac{3}{2}$.

7. Possible extensions. Just as Algorithm A_{max} gives a better bound than Algorithm A at the expense of running more trials, it seems possible to improve the bound on Algorithms B and B* by a similar approach, giving Algorithm B_{max} and B^*_{max}. In addition, there are factors not considered, which means that the bound on Algorithms B and B* is pessimistic. One such factor is mentioned above, namely, that the cuts in T_m may have many more than m edges. Another such factor is the interdependence of the behavior of different cuts. In the above analysis, we have assumed the worst possible case, namely, that at most one cut C has a high absolute capacity per edge at a time. Thus the probability that some cut C has a high absolute capacity per edge is at most the sum of the probabilities for different cuts C. In reality, the chances that two cuts C_1 and C_2 will have a high absolute capacity per edge, are closely related much of the time. Note, for example, that $|\Delta_{C_1} - \Delta_{C_2}|$ is bounded by $|Y_1 - Y_2|$ for Algorithm A or $K|Y_1 - Y_2|$ for Algorithm B, where C_1 *is* (Y_1, Z_1) and C_2 is (Y_2, Z_2). Therefore cuts that are similar to each other will behave similarly. In fact, the expected absolute difference between the number of sources and sinks in $|Y_1 - Y_2|$ is on the order of $\sqrt{|Y_1 - Y_2|}$, or $\sqrt{K|Y_1 - Y_2|}$ for Algorithm B. This means that the average correlation between cuts is much higher than the worst-case value. That is, the expected value of $|\Delta_{C_1} - \Delta_{C_2}|$ is on the order of $\sqrt{K|Y_1 - Y_2|}$ for Algorithm B. The effect of all this is to reduce the effective number of cuts in G, improving the performance of Algorithms B and B*. Since Algorithms A, A_{max}, and A* are based on worst-case bounds, and not on independence assumptions, those bounds are not affected.

Also, for many graphs, it seems that the edges in an optimal cut will be more likely to be included in the cuts found by Algorithm A than the other edges in the graph. So, a promising approach is to run Algorithm A many times (say 100 times), and with each edge keep the number of times it is included in the cut found by Algorithm A. Then, find the minimium value t such that the edges found t or more times, disconnect the graph, and let this set of edges be the cut output by the algorithm. This approach is fast and also seems likely to obtain near optimal cuts for many graphs. However, we have not analyzed this approach at all.

Acknowledgments. This problem arose from discussions with Arny Rosenberg and Kye Hedlund at Microelectronics Center of North Carolina concerning VLSI design algorithms.

REFERENCES

T. BUI, S. CHAUDHURI, T. LEIGHTON, AND M. SIPSER [1984], *Graph bisection algorithms with good average case behavior*, in Proc. 25th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 181-192.

L. R. FORD, JR. AND D. R. FULKERSON [1962], *Flows in Networks*, Princeton University Press, Princeton, N J.

M. R. GAREY, D. S. JOHNSON, AND L. STOCKMEYER [1976], *Some simplified* NP*-complete graph problems*, Theoret. Comput. Sci., 1, pp. 237-267.

N. KARMARKAR [1984], *A new polynomial-time algorithm for linear programming*, Combinatorica, 4, pp. 373-395.

T. LEIGHTON AND S. RAO [1988], *An approximate max-flow min-cut theorem for uniform multicommodity flow problems with applications to approximation algorithms*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC.

R. J. LIPTON AND R. E. TARJAN [1977], *Application of a planar separator theorem*, in Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 162–170.

——— [1979], *A separator theorem for planar graphs*, SIAM J. Appl. Math., 36, pp. 177–189.

K. MEHLHORN [1984], *Data Structures and Algorithms* 2: *Graph Algorithms and* NP-*Completenes*, Springer-Verlag, New York.

G. MILLER [1984], *Finding small simple cycle separators for* 2*-connected planar graphs*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, pp. 376–382.

S. RAO [1987], *Finding near optimal separators in planar graphs*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, pp. 225–237.

S. RAJASEKARAN AND J. REIF [1988], *Nested annealing*: *A provable improvement to simulated annealing*, ICALP, Finland, July, 1988.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 281-296, April 1990

006

ON THE EFFICIENT GENERATION OF LANGUAGE INSTANCES*

LAURA A. SANCHIS† AND MARK A. FULK‡

Abstract. Polynomial-time Turing machines that output instances of a given language are considered, where the instances are required to have a certain length specified by the input. Two types of generating machines are investigated. The first, called a *constructor*, is deterministic and outputs one string in the language having the specified input length, if such a string exists. A *generator* is nondeterministic and may output different strings in the language using different computations on the same input; it is required, however, that for any string in the language satisfying the input constraint, there be some computation of the generator on this input that produces the string. Although most P and NP languages examined appear to have such polynomial-time constructors and generators, it is shown that the question of whether all NP languages have such machines is related to other open questions in complexity theory and that even under the assumption that P is not equal to NP, the question cannot be resolved using techniques that relativize.

More general and/or flexible types of generators are also considered; namely, generators based on other parameters besides length; generators that are not necessarily capable of outputting all of the instances in the language satisfying the input constraint; and generators where the output instance does not need to satisfy the input constraint exactly. Several results are proved about the existence of such generators for various types of languages.

Key words. generation, construction, string length, sparse languages, NP, oracles

AMS(MOS) subject classifications. 68Q15, 68Q05

1. Introduction.

Consider a well-known language in NP, such as the set of Hamiltonian graphs in a suitable language encoding. Given a length, it is straightforward to determine whether or not there exists a string in this language having that particular length. Also it is quite easy to construct such a string if it exists: after determining n and m such that a graph with n vertices and m edges has an encoding with the given length, and such that $m \geqq n$, simply construct a Hamiltonian cycle and then add extra edges if necessary. With a little more thought it is also possible to come up with an efficient (polynomial-time) nondeterministic or probabilistic procedure that can output all Hamiltonian graphs having a given encoding length (or equivalently, having a given number of vertices and edges), although this procedure will probably include repetitions and will not necessarily output the graphs with uniform probabilities. In this paper we investigate whether such efficient construction or generation procedures exist for all languages in P or NP. It turns out that most (although not all) well-known languages in NP appear to have such construction and generation algorithms, but that determining whether this is true for all NP languages is a hard problem and cannot be resolved without answering some major open question in complexity theory.

There are several ways in which a machine that generates instances of a language can be defined. The various factors to be considered include the following:

- How many instances are generated on each run; that is, does the machine output one, some, or all of the instances of the language that satisfy the input constraint each time it is run?

* Received by the editors October 31, 1988; accepted for publication (in revised form) August 11, 1989.

† Computer Science Department, University of Rochester, Rochester, New York 14627. Present address, Computer Science Department, Colgate University, Hamilton, New York 13346. This work was done while this author was an AT&T Bell Labs Scholar.

‡ Computer Science Department, University of Rochester, Rochester, New York 14627. This work was supported in part by National Science Foundation Coordinated Experimental Research grant DCR-8320136.

- How tightly does the input specification control the properties of the output? For instance, if the input is a specified length, does an output instance have to have exactly the length specified, or can it be within some given range of the input length?
- Should the machine be deterministic, nondeterministic, or should it be required to output instances uniformly or with a given probability distribution?

Several of the models of generation that can be obtained by various answers to the above questions have theoretical interest and/or practical applications. In this paper we have chosen to investigate generation models where the machine outputs one instance of the language each time it is run. We also impose no distribution requirements on the generators. We look at two types of generating machines. The first, called a *constructor*, is deterministic. Given an input, it outputs exactly one string in the language having the specified input length, if such a string exists. A *generator* is nondeterministic and may output different strings in the language using different computations on the same input. Furthermore, we require that for any string in the language satisfying the input constraint, there be some computation of the generator on this input that produces the string.

Other models of generation complexity have been studied in the literature. Jerrum, Valiant, and Vazirani in [17] use the concept of p-relations (relations that can be checked in polynomial time) to model the construction and uniform generation of solutions for given problem instances. Uniform language instance generation for languages in P is a special case of the generation considered in [17], since instance generation of a language L can be modeled by the relation R_L, where xR_Ly if and only if x specifies a length and y is a string in L having length x. Thus it follows from results in [17] that if L is any language in P, it is possible to generate uniformly in polynomial time all instances of L of a given length, using a probabilistic Turing machine equipped with a $\#\mathrm{P}$ oracle or a Σ_2^p-oracle. Note that it is easy to see that construction (and in fact existence) can be done in polynomial time for all p-relations if and only if $\mathrm{P} = \mathrm{NP}$. Such a result is not at all apparent, however, in the context of language instance generation. If $\mathrm{P} = \mathrm{NP}$, then it can indeed be shown that all P and NP languages must have polynomial-time constructors and (nonuniform) generators. However, we will show that the converse cannot be proved using techniques that relativize.

Another model of language instance generation that is related to cryptography and interactive proof systems is investigated by Abadi et al. [1], Hemachandra et al. [14], and Feigenbaum, Lipton, and Mahaney [9]. These papers investigate the existence of *invulnerable* generators for languages in NP. If L is a language accepted by machine M, then an *α-invulnerable generator* for L is a machine that on input 1^n generates pairs $\langle x, w\rangle$, where $x \in L$, $|x| = n$, and w is an accepting computation for x in M; moreover the pairs are generated according to a distribution under which any polynomial-time adversary who is given x fails to find an accepting computation for x, with probability at least α, for infinitely many lengths n. The emphasis in [1], [14], [9] is on the hardness of the distribution generated.

Other work dealing with the generation in polynomial time of specific structures, such as graphs having certain properties, may be found in [28], [2], [26]. See also related work in [15].

This paper is organized as follows. Section 2 contains preliminary definitions and notation. Section 3 presents some results on polynomial-time generation with oracles. Section 4 presents the main results about the existence of polynomial-time constructors and generators for languages in P and NP. Section 5 examines the properties of polynomial-time generators when viewed as NP machines. Section 6 deals with the

existence of polynomial-time constructors for languages in the polynomial-time hierarchy. In § 7 we examine other more general and less restrictive definitions of generators, and we give several results about these types of generators and their relation to the generation model studied in the rest of the paper. Section 8 contains relativization results pertaining to the existence of constructors and generators for languages in P, NP, and co-NP.

2. Preliminaries. Our Turing machine (TM) model is defined as in [16]. A Turing machine may be deterministic (DTM) or nondeterministic (NDTM). A Turing machine that (when it halts) either accepts or rejects its input is called an *acceptor.* A Turing machine that, in addition to the input and work tapes, has an output tape is called a *transducer.* An *oracle* Turing machine (OTM) is also defined as in [16].

The language classes in the polynomial-time hierarchy are defined as in [25]. We let $E = \bigcup_{c \geq 0} \mathrm{DTIME}(2^{cn})$, and $\mathrm{NE} = \bigcup_{c \geq 0} \mathrm{NTIME}(2^{cn})$.

A nondeterministic Turing machine acceptor M is *categorical* if for each x there exists at most one computation of M on input x that leads to acceptance. The class of languages in NP that are accepted by categorical NP machines is called UP [27]. We have $\mathrm{P} \subseteq \mathrm{UP} \subseteq \mathrm{NP}$; it is not known whether these inclusions are proper.

The class D^p consists of all languages that are intersections of a language in NP and a language in co-NP [21]. $\mathrm{NP} \subseteq \mathrm{D}^p$, $\text{co-NP} \subseteq \mathrm{D}^p$, and $\mathrm{NP} = \mathrm{D}^p$ if and only if NP = co-NP.

A language L is *sparse* if the number of strings in L of length n is at most $p(n)$, for each $n \geq 1$, for some polynomial p. A *tally* language is a language over a one-letter alphabet. A language is P-*printable* if there exists a polynomial-time procedure that given input 1^n outputs all strings in the language of length n. Note that if a language is P-printable it must be sparse. An infinite language is P-*immune* if it has no infinite subset in P.

We denote the set of natural numbers by N and the set of positive integers by N^+. Some of the proofs in the paper make use of the pairing function $\rho(n, m) = (n+m)(n+m+1)/2+n$. This function is polynomial-time computable, is one-to-one and onto from $N \times N$ to N, and it has the property that $\rho(n, m) \geq nm$ for all $n, m \in N$.

We assume without loss of generality that NP machines make only two-way nondeterministic branches and that the polynomial bounding the computation time is of the form n^k for some integer k. Given an NP machine M with time bound n^k, we say a string of length n^k codes a computation of M on an input of length n, if the bits of the string encode the nondeterministic branches that M makes in the computation (with extra padding 0's allowed).

3. Generation with oracles.

DEFINITION 3.1. A nondeterministic oracle Turing machine transducer equipped with an oracle A *generates* a language L if on input 1^n and querying oracle A, it outputs a string in L of length n, if such a string exists, and outputs Λ otherwise. Moreover, for each string x of length n in L there should be at least one computation of the machine on input 1^n that produces x.

DEFINITION 3.2. Let L be a language. The *padded prefix closure* of L is defined as

$$\tilde{L} = \{x\#^m \mid \exists y \text{ such that } xy \in L \text{ and } |y| = m\}.$$

PROPOSITION 3.1. (1) *A language L can be generated by a nondeterministic polynomial-time Turing machine equipped with an oracle for $\tilde{L}$.*

(2) *For $k \geq 1$, if $L \in \Sigma_k^p$, L can be generated by a nondeterministic polynomial-time Turing machine equipped with a Σ_k^p-oracle.*

Proof. (1) Given input 1^n, a generator for L constructs a string of length n in L (if one exists) bit by bit, consulting the oracle at each step to determine how to add the next bit in order to form a valid prefix of a string of length n in L.

(2) If $L \in \Sigma_k^p$ for $k \geq 1$, then $\tilde{L}$ is also in Σ_k^p. The result then follows from part (1). □

The above proof uses part of a technique that was used in [17] to show that uniform generation for p-relations can be done by a polynomial-time machine equipped with a #P oracle.

It follows from Proposition 3.1 that (nonuniform) generation for NP languages can be placed within the second level of the polynomial-time hierarchy. A major question investigated by this paper is whether generation for NP languages can in fact be placed in the first level of the hierarchy. Note that in [17] it is shown that uniform generation for p-relations can be placed within the third level of the polynomial-time hierarchy.

4. Construction and generation for P and NP languages.

DEFINITION 4.1. (1) A *polynomial-time detector* (*PTD*)[1] for a language L is a *deterministic* Turing machine transducer that runs in polynomial time and that on input 1^n outputs 1 if there exist strings of length n in L and outputs 0 otherwise.

(2) A *polynomial-time constructor* (*PTC*) for a language L is a *deterministic* Turing machine transducer that runs in polynomial time and that on input 1^n outputs a string in L of length n, if such a string exists, and outputs Λ otherwise.

(3) A *polynomial-time generator* (*PTG*) for a language L is a *nondeterministic* Turing machine transducer that runs in polynomial time and that on input 1^n outputs a string in L of length n, if such a string exists, and outputs Λ otherwise. Moreover, for each string $x \in L$ of length n there should exist some computation of the generator on input 1^n that outputs x.

(4) A *categorical PTG* for a language L is a PTG for L such that, for each string $x \in L$ of length n, there is exactly one computation of the generator on input 1^n that outputs x.

LEMMA 4.1. (1) *If a language has a* PTG *it has a* PTC; *if a language has a* PTC *it has a* PTD.

(2) *If a language has a* PTG *it is in* NP.

(3) *A language in* NP *has a* PTC *if and only if it has a* PTG.

Proof. (1) and (2) are clear from the definitions. To show (3), let L be in NP and have PTC C_L. A PTG G_L for L may be defined as follows. On input 1^n G_L nondeterministically constructs a string z of length n. It then runs a polynomial-time NDTM acceptor for L on input z. If this machine accepts, then G_L outputs z. Otherwise, G_L simulates C_L on input 1^n and outputs the string produced by C_L (or Λ if C_L outputs Λ). □

Note that the argument in the above proof cannot be used to show that all NP languages have PTGs, because a PTG can output Λ only if there are no strings of the specified length in the language.

Since only languages in NP can have PTGs, a natural question to ask is whether in fact all languages in NP have PTGs. If not, do at least all languages in P have PTGs? By part (3) of the above lemma this is equivalent to asking whether or not all languages in P or NP have PTCs. We will show that, although these questions are hard to answer, to determine whether or not all NP languages have PTDs, PTCs, or

[1] A detector is called a *tallier* in [24].

PTGs, it suffices to answer the question for languages in P. The same pattern is repeated for other types of generating machines dealt with later in this paper.

As mentioned in the introduction, it appears fairly easy to construct PTDs, PTCs, and PTGs for most well-known languages in P and NP. As an example, consider the NP-complete language corresponding to the 3-satisfiability (3SAT) problem [10]. Each instance of this problem consists of a set of variables $\{x_1, \ldots, x_n\}$ and a set of clauses $\{c_1, \ldots, c_m\}$, where each clause consists of the logical disjunction of three literals (each literal is of the form x_i or $\bar{x}_i$ for some i). The resulting formula is satisfiable if there is a truth assignment for the variables that simultaneously makes every clause true. Let 3SAT denote the NP-complete language consisting of encodings of all such satisfiable formulas. We assume that the length of the string encoding each instance is polynomially related to the parameters n and m, where n denotes the number of variables and m the number of clauses. To construct an instance of 3SAT with n variables and m clauses, randomly assign a truth value T or F to each of $x_1, x_2, \cdots, x_n$. Let $u_i = x_i$ if x_i was assigned T, $u_i = \bar{x}_i$ otherwise. To form each of the m clauses first randomly choose some u_i, thus ensuring that the clause is true, and then randomly choose any two more variables for the clause from among $x_1, x_2, \cdots, x_n, \bar{x}_1, \bar{x}_2, \cdots, \bar{x}_n$. Clearly each satisfiable formula with n variables and m clauses is generated in this manner. Moreover the number of times a particular satisfiable formula is generated is proportional to the number of different assignments that satisfy it.

There are, however, languages in NP for which no PTC is known. Such a language is the set of prime numbers (encoded in binary notation) [22]. Nevertheless there does exist a trivial PTD for the set of prime numbers, since for sufficiently large n there always exists a prime of length n (Bertrand's postulate, see [19]).

4.1. Polynomial-time detectors (PTDs).

PROPOSITION 4.1. *The following are equivalent*:

(1) *All tally languages in* NP *have* PTDs.

(2) *All languages in* NP *have* PTDs.

(3) *All languages in* P *have* PTDs.

(4) *There are no tally languages in* NP-P.

Proof. For any language L in NP, let $L_T = \{1^n \mid \exists x \in L \text{ with } |x| = n\}$. Note that L has a PTD if and only if L_T has a PTD. Thus (1)$\Leftrightarrow$(2). Also it is clear that a tally language is in P if and only if it has a PTD. Hence we have (1)$\Leftrightarrow$(4). It is also clear that (2)$\Rightarrow$(3).

To complete the proof it will be shown that (3)$\Rightarrow$(4). Let L be any tally language in NP and let M be a polynomial-time NDTM that accepts L and that runs in time bounded by n^k on inputs of length n. Define the language L_S to consist of all strings x of length n^k such that, if M is given input 1^n and if M follows the computation coded in x, then M accepts. L_S is in P, and hence by assumption L_S has a PTD. Note 1^n is in L if and only if L_S has a string of length n^k, and thus L is in P. □

4.2. Polynomial-time constructors (PTCs) and polynomial-time generators (PTGs). Book shows in [5] that there are no tally languages in NP-P if and only if $E = NE$, while Hartmanis shows in [13] that there are no sparse languages in NP-P if and only if $E = NE$. Thus it follows from the previous proposition that if all sparse NP languages have PTCs (or PTDs), then there are no sparse languages in NP-P. (See also [12].)

If we assume that there are no sparse languages in NP-P, and if L is any sparse language in NP, then its padded prefix closure $\tilde{L}$ is also sparse and in NP, and hence in P. From the argument in the proof of Proposition 3.1 it follows that L has a PTC

and in fact is P-printable. Clearly also if a language is P-printable it has a PTC. We therefore have the following corollary of Proposition 4.1.

COROLLARY 4.2. *All sparse languages in* NP *have* PTCs (PTDs) *if and only if all sparse languages in* NP *are* P*-printable.*

The above corollary can also be derived from a more direct argument (bypassing the $E = NE$ connection). Assume all sparse languages in NP have PTCs and let L be any sparse language in NP, with $p(n)$ bounding the number of strings of length n in L, where p is a polynomial. Let the language L' contain, for each n, a single string of length $\rho(n, p(n))$, consisting of the concatenation, in lexicographic order, of all strings of length n in L, padded with extra $*$ symbols ($* \notin \Sigma$) to achieve the required length. Clearly L' is sparse, and we can also show that L' is in NP, as follows. Define $L_E = \{1^{\rho(n,m)} \mid \exists$ at least m strings of length n in $L\}$. L_E is a tally language in NP. By our assumption L_E must have a PTC, implying that it is in P, and therefore L' is in NP. Therefore by assumption L' has a PTC, and it immediately follows that L is P-printable. □

It is an open question where the converse of the last corollary is true, or equivalently, whether if all languages in NP have PTDs then they all must have PTCs. However, we can show the following.

THEOREM 4.3. *If there are no sparse languages in* $D^P - P$*, then all languages in* NP *have* PTCs.

Proof. Let L be a language in NP and consider the language L_F consisting for each n of the lexicographically first string of length n in L, if such a string exists. This language is sparse and is in D^P. Its padded prefix closure $\tilde{L}_F$ is also sparse and in D^P. Hence if there are no sparse languages in $D^P - P$, $\tilde{L}_F$ is in P and so by Proposition 3.1 L_F has a PTG, and hence also a PTC. This PTC is also a PTC for L. □

Thus if all NP languages have PTCs, then there are no sparse languages in NP-P. On the other hand, if there exists some NP language that does not have a PTC, then there is a sparse language in $D^P - \mathrm{P}$ (implying of course that $\mathrm{P} \neq \mathrm{NP}$). Hence one cannot hope to resolve the question of whether or not all NP languages have PTCs without resolving some major open question in complexity theory.

There are certain languages that are particularly easy to generate, namely, those languages whose padded prefix closures are in P. In fact it follows from the proof of Proposition 3.1 that all such languages have categorical PTGs.

DEFINITION 4.2. The language class *Prefix-P* consists of all those languages whose padded prefix closures are in P.

Note that Prefix-P $\subseteq$ P.

PROPOSITION 4.4. *If a language is in Prefix-*P *it has a categorical* PTG.

PROPOSITION 4.5. *Prefix-*P = P *if and only if* $P = NP$.

Proof. Consider the language post-3SAT consisting of all strings of the form $y \# x$, such that y is a satisfiable 3SAT formula and x is a truth assignment satisfying y. It is clear that post-3SAT is in P, but post-3SAT cannot be in Prefix-P unless P = NP. □

If $\mathrm{P} \neq \mathrm{NP}$, the class of languages that have PTGs is not restricted to those languages in Prefix-P. For instance, 3SAT and post-3SAT have PTGs even though, unless P = NP, neither of these languages is in Prefix-P. However, any language that has a PTG can be characterized as the image of another language in Prefix-P under a polynomial mapping of a certain type.

DEFINITION 4.3. A language L is *prefixable* if there exists a language L' in Prefix-P, a polynomial f that is one-to-one on the positive integers, and a polynomial-time computable onto function $g: L' \to L$ such that $|x| = f(|g(x)|)$ for all $x \in L'$.

THEOREM 4.6. *A language has a* PTG *if and only if it is prefixable.*

Proof. Suppose L is prefixable, and let L', f, g have the required properties as stated in the definition. Since L' is in Prefix-P, it has a PTG $G_{L'}$. A PTG G_L for L can then be defined as follows. On input 1^n, G_L computes $f(n)$ and simulates $G_{L'}$ on input $1^{f(n)}$. If $G_{L'}$ outputs Λ, then G_L outputs Λ. If $G_{L'}$ outputs a string x, then G_L outputs $y = g(x)$.

Suppose conversely that L has a PTG G_L that runs in time bounded by n^k on input 1^n. Let L' consist of all strings of the form $x \# y$ ($\# \notin \Sigma$), where $n \geq 1$, $|y| = n$, $|x| = n^k$, and if G_L follows the computation coded in x, G_L outputs y. Then L' is in Prefix-P (recall that a computation of G_L can output Λ on input 1^n only if there are no strings of length n in L). Thus we can define $f(n) = n + 1 + n^k$, and $g: L' \to L$ by $g(x \# y) = y$. □

The following theorem shows that to decide whether or not all NP languages have PTGs, it is sufficient to resolve the question for languages in P.

THEOREM 4.7. *All languages in* NP *have* PTGs *if and only if all languages in* P *have* PTGs.

Proof (sketch). The same technique that was used in the last proof can be used to prove this theorem. Let L be any language in NP, recognized by NDTM M, with time bound n^k. Let L' consist of all strings of the form $x \# y$, where $n \geq 1$, $|y| = n$, $|x| = n^k$, and if M on input y follows the computation coded in x, M accepts. Then L' is in P, and if L' has a PTG, it is not hard to see that L must also have a PTG. □

Finally, another fact that has bearing on the question of whether or not all languages in NP have PTGs is the existence of a language in NP that is complete in this regard.

THEOREM 4.8. *There exists a language K in* NP *such that K has a* (*categorical*) PTG *if and only if all languages in* NP *have* (*categorical*) PTGs.

Proof. The proof uses the same technique that was used in the second proof of Corollary 4.2. Let $N_1, N_2, \cdots$, be an effective enumeration of all of the NP Turing machine acceptors. Let machine N_i have time bound p_i, where each p_i is a one-to-one polynomial. For each pair of positive integers i, n, code each string of length n accepted by N_i into a string of length $\rho(i, p_i(n))$, padding the string with $*$'s to achieve the required length. Let the language K consist of all strings formed in this way from all such pairs. Note that K is just the standard universal NP-complete language coded into strings of special lengths. It is not hard to see that if K has a PTG, then all NP languages have PTGs. □

COROLLARY 4.9. *All* NP *languages have* (*categorical*) PTGs *if and only if all* NP-*complete languages have* (*categorical*) PTGs.

4.3. Categorical and lexicographic polynomial-time generators (PTGs). Not only do most well-known NP languages have PTGs, but many also have PTGs with the property that the number of times each string is produced is equal to the number of accepting computations for the string in some NP acceptor for the language. An interesting question is whether or not all NP languages have such generators.

DEFINITION 4.4. A language L is *machine-categorically generable* if for every NP machine M that accepts L, there exists a PTG G_M for L such that the number of times each string x in L of length n is generated by G_M on input 1^n equals the number of accepting computations for x in M.

If a language is in P, it has a deterministic polynomial-time acceptor, and the PTG corresponding to this acceptor according to the above definition would be categorical. Thus if we restrict our inquiry to languages in P and to the deterministic polynomial-time machines accepting these languages, then the question becomes

whether or not all languages in P have categorical PTGs. Again it turns out that to answer the question about NP languages, it suffices to answer the question for languages in P, as is shown by the following theorem.

THEOREM 4.10. *All languages in P have categorical* PTGs *if and only if all languages in* NP *are machine-categorically generable.*

The proof of this theorem uses the same argument used in the proofs of Theorems 4.6 and 4.7. (See [24] for details.)

In the following definition, we assume all computations of a PTG with time bound n^k on input 1^n are described with strings of length n^k, padded with 0's if necessary.

DEFINITION 4.5. Let G be a PTG for a language L with time bound n^k on input 1^n. G is lexicographic if for all x, y, w, z such that $x, y \in L$, $|x| = |y| = n$, $|w| = |z| = n^k$, and $w \leqq z$, if G on input 1^n and following the computation coded in w outputs x, and G on input 1^n and following the computation coded in z outputs y, then $x \leqq y$.

PROPOSITION 4.11. *Let L be a language. The following are equivalent*:

(1) *L is in* Prefix-P.

(2) *L has a lexicographic* PTG.

(3) *The language* $L_{int} = \{x \sqcup y \mid |x| = |y|, \exists z \in L \text{ such that } x \leqq z \leqq y\}$ *is in* P.

Proof. We will show that (1)$\Rightarrow$(2)$\Rightarrow$(3)$\Rightarrow$(1). (1)$\Rightarrow$(2) follows from the proof of Proposition 3.1. (3)$\Rightarrow$(1) is easy, as determining whether $x\#^m$ is a padded prefix of a string in L is equivalent to asking whether $x0^m \sqcup x1^m$ is in L_{int}.

To prove that (2)$\Rightarrow$(3), let G be a lexicographic PTG for L, where G runs in time bounded by n^k on input 1^n. We may construct another PTG G' for L that always makes exactly n^k nondeterministic branches in its computations and still has the lexicographic property. To determine whether $x \sqcup y$ is in L_{int}, where $|x| = |y| = n$, perform a binary search on the set of strings of length n^k, to determine whether there exists a string w of length n^k such that if G' is run on input 1^n following the computation coded in w, then G' outputs a string z such that $x \leqq z \leqq y$. This can be done in time $O((\log 2^{n^k})n^k) = O(n^{2k})$. □

Another interesting question is whether or not all P or NP languages have lexicographic PTGs. As is the case for ordinary PTGs, it suffices to answer this question for languages in P. In this case the question turns out to be equivalent to whether P = NP. This is because by the above proposition all languages in P have lexicographic PTGs if and only if P = Prefix-P, which is equivalent to P = NP.

COROLLARY 4.12. P = NP *if and only if all languages in* P *have lexicographic* PTGs.

5. Some properties of polynomial-time generators (PTGs). The results of this section show that even should it be the case that not all NP machines have PTGs, PTGs are rather diversified, both with regard to their properties and with regard to the types of languages that have PTGs.

THEOREM 5.1. (1) *If* $P \neq NP$, *then there exists a* PTG *for a language in* NP-P.

(2) *If* P $\neq$ UP, *then there exists a categorical* PTG *for a language in* UP-P.

Proof. (1) This follows from the fact that many NP-complete languages, e.g., 3SAT, have PTGs.

(2) Let L be a language in UP-P, where L is recognized by a categorical machine M that makes exactly n^k moves on inputs of length n. Define the language $L' = L_1 \cup L_2$. L_1 consists of all strings of the form $x\#y$, where $|x| = n$, $|y| = n^k$, and if M is run on input x following the computation coded in y, M rejects. L_2 consists of all strings of the form $x*^{n^k+1}$, where $x \in L$. Thus L' is in UP-P, and L' has a categorical PTG: on input 1^m, where $m = n + 1 + n^k$, this PTG randomly constructs strings x and y of lengths

n and n^k, respectively, and runs M on input x following the computation coded in y. If M rejects, the PTG outputs $x\#y$; otherwise, it outputs $x*^{n^k+1}$. □

DEFINITION 5.1. An NP machine for a language L is *traceable* if there exists a polynomial-time procedure that on input $x \in L$ outputs an accepting computation of the machine on input x.

A PTG for a language L can be used as an NP recognizer for L. In terms of PTGs viewed as NP machines, traceability has the following meaning: a PTG for a language L is traceable if there exists a polynomial-time procedure that on input $x \in L$ outputs some computation of the PTG that on input 1^n outputs x. Clearly any language that has a traceable PTG must be in P, but it is not necessarily true that all PTGs for languages in P must be traceable. It is not hard to see, however, that if a language in P has a PTG it has a traceable PTG (using the argument in the proof of Lemma 4.1).

Borodin and Demers show in [6] that if $P \neq NP \cap$ co-NP there is an NP machine for Σ^* that is not traceable. Similarly it follows from results by Grollmann and Selman [11] that if $P \neq UP \cap$ co-UP, then there is a UP machine for a language in P that is not traceable. These hypotheses also imply the existence of nontraceable PTGs, as seen in the following theorem.

THEOREM 5.2. (1) $P = NP$ *if and only if all* PTGs *are traceable.*

(2) $P = UP$ *if and only if all categorical* PTGs *are traceable.*

(3) $P = UP \cap$ co-UP *if and only if all categorical* PTGs *for languages in* P *are traceable.*

(4) *If* $P \neq NP \cap$ co-NP, *then there exists a* PTG *for a language in* P *that is not traceable.*

Proof. The ($\Rightarrow$) implications in (1), (2), and (3) are true for all NP machines, not just PTGs (see proofs of similar results in [11]). The ($\Leftarrow$) implications in (1) and (2) follow from Theorem 5.1.

To prove the ($\Leftarrow$) implication in (3), let L be a language in $(UP \cap \text{co-UP}) - P$. Let M_1, M_2 be categorical machines accepting L and $\bar{L}$, respectively, where n^k bounds the number of moves made by both M_1 and M_2 on inputs of length n. Define languages L_1 and L_2 as follows. $L_1 = U_1 \cup V_1$. U_1 consists of all strings of the form $x\#y$, where $|x| = n$, $|y| = n^k$, and M_1 on input x and following the computation coded in y rejects. V_1 consists of all strings of the form $x*^{n^k+1}$, where $|x| = n$ and $x \in L$. $L_2 = U_2 \cup V_2$. U_2 consists of all strings of the form $x\%y$, where $|x| = n$, $|y| = n^k$, and M_2 on input x and following the computation coded in y rejects. V_2 consists of all strings of the form $x*^{n^k+1}$, where $|x| = n$ and $x \in \bar{L}$. As seen in the proof of Theorem 5.1, L_1, L_2 have categorical PTGs. $L_1 \cap L_2$ is empty, hence a categorical PTG G can be constructed for $L_1 \cup L_2$ whose first branch point consists of deciding whether to output a string in L_1 or L_2. It is evident that $L_1 \cup L_2$ is in P. However, G cannot be traceable for then L would be in P.

(4) Let L be a language in $(NP \cap \text{co-NP}) - P$. Define the language L' as follows: $L' = \{1x \mid x \in L\} \cup \{00y \mid y \in \Sigma^*\}$. So $\bar{L}' = \{1x \mid x \in \bar{L}\} \cup \{01y \mid y \in \Sigma^*\}$. L', $\bar{L}'$ are both in NP and have trivial PTCs; hence they have PTGs. We may then define a PTG for Σ^* whose first branch point consists of deciding whether to output a string in L' or $\bar{L}'$. This PTG cannot be traceable, since that would imply that L is in P. □

6. Construction for languages in the polynomial-time hierarchy. This section contains results about the existence of PTCs for language classes in the polynomial-time hierarchy. These results are generalizations of results proved earlier for languages in P and NP.

THEOREM 6.1. *All languages in* Δ_k^p *have* PTCs *if and only if all languages in* Σ_k^p *have* PTCs.

DEFINITION 6.1. If L is any language, let L_ρ denote the language consisting of all strings formed by taking the concatenation of any sequence of m distinct strings of length n from L, and padding the resulting string with $*$'s, to form a string of length $\rho(n, m)$.

PROPOSITION 6.2. *Let* $\mathscr{C}$ *be any class of languages such that, if* $L \in \mathscr{C}$, *then* L_ρ *is also in* $\mathscr{C}$. *Then if all languages in* $\mathscr{C}$ *have* PTCs, *then all sparse languages in* $\mathscr{C}$ *are* P-*printable.*

Proof. Let L be any sparse language in $\mathscr{C}$ and let $p(n)$, where p is a polynomial, bound the number of strings of length n in L. Since L is in $\mathscr{C}$, L_ρ is in $\mathscr{C}$, and by assumption L_ρ has a PTC C. To print out all of the strings of length n in L in polynomial time, simulate C on inputs $\rho(n, p(n))$, $\rho(n, p(n)-1)$, $\cdots$, etc., until a string other than Λ is produced. This string must contain all strings of length n in L. □

Note that for any $k \geqq 0$, the language classes Δ_k^p, Σ_k^p, and Π_k^p all satisfy the hypothesis of the above proposition. We therefore have the following.

COROLLARY 6.3. (1) *If all languages in* Δ_k^p *have* PTCs, *then all sparse languages in* Δ_k^p *are* P-*printable.*

(2) *If all languages in* Σ_k^p *have* PTCs, *then all sparse languages in* Σ_k^p *are* P-*printable.*

(3) *If all languages in* Π_k^p *have* PTCs, *then all sparse languages in* Π_k^p *are* P-*printable.*

Part (2) of the above corollary can be strengthened as follows. The proof of this theorem is similar to the second proof of Corollary 4.2.

THEOREM 6.4. *All sparse languages in* Σ_k^p *have* PTCs *if and only if all sparse languages in* Σ_k^p *are* P-*printable.*

The next theorem is a generalization of Theorem 4.3.

DEFINITION 6.2. For $k \geqq 1$, define $\mathrm{Diff}_k = \{L_1 - L_2 \mid L_1, L_2 \in \Sigma_k^p\}$. Note $\mathrm{Diff}_1 = D^p$.

THEOREM 6.5. *If there are no sparse languages in* $\mathrm{Diff}_k - P$, *then all languages in* Σ_k^p *have* PTCs.

7. Other types of generators. To distinguish the generators defined in earlier sections from the different types of generators introduced in this section, the former generators will be referred to as *length-restricted* generators.

7.1. Parameter-based generation.

DEFINITION 7.1. Let $l_1, \cdots, l_k$ be polynomial-time computable functions, $l_j : \Sigma^* \to N$ for $1 \leqq j \leqq k$, and q, r be polynomials such that $l_j(x) \leqq q(|x|)$ for $1 \leqq j \leqq k$ and $|x| \leqq r(l_1(x), \cdots, l_k(x))$ for all x. An $(l_1, \cdots, l_k)$-PTG for a language L is a polynomial-time NDTM M that on input $1^{v_1}\#1^{v_2}\#\cdots\#1^{v_k}$, either outputs a string x in L such that for $1 \leqq j \leqq k$, $l_j(x) = v_j$, or outputs the symbol Λ indicating that no such string exists. Furthermore for every string x in L such that $l_j(x) = v_j$ for $1 \leqq j \leqq k$ there exists some computation of M on input $1^{v_1}\#1^{v_2}\#\cdots\#1^{v_k}$ that outputs x.

THEOREM 7.1. *Let* $(l_1, \cdots, l_k)$, q, r *be as in the above definition. If all languages in* NP *have length-restricted* PTGs, *then all languages in* NP *have* $(l_1, \cdots, l_k)$-PTGs.

Proof. Suppose all languages in NP have length-restricted PTGs, and let L be any language in NP. By repeated compositions of ρ it is possible to define a polynomial function $\rho_{k+1} : N^{k+1} \to N$ on $k+1$ variables that is one-to-one and for which $\rho_{k+1}(n_1, \cdots, n_{k+1}) \geqq n_1 \cdots n_{k+1}$ for all $n_1, \cdots, n_{k+1} \in N$. For each $x \in L$, let $f(x) = \rho_{k+1}(l_1(x)+1, \cdots, l_k(x)+1, r(l_1(x), \cdots, l_k(x))+1)$ and let $s(x)$ be the string of length $f(x)$ consisting of the concatenation of x and $(f(x) - |x|)$ padding $(*)$ symbols. Define $S = \{s(x) \mid x \in L\}$. S is clearly in NP, so S has a PTG G_S, which can be used to define an $(l_1, \cdots, l_k)$-PTG G_L for L as follows. On input $1^{v_1}\#\cdots\#1^{v_k}$, G_L computes

$m = \rho_{k+1}(v_1+1, \cdots, v_k+1, r(v_1, \cdots, v_k)+1)$ and simulates G_S on input 1^m. G_S then outputs a string y of length m in S (if such a string exists); this string will contain as a prefix a string $x \in L$ with parameter values $v_1, \cdots, v_k$. □

7.2. Semi-polynomial-time generators (semi-PTGs). A PTC outputs at most one string of each length from its associated language. A PTG, on the other hand, must be capable of outputting (in some nondeterministic computation) each string in the language. It is possible to define a type of machine that in generating power lies between a PTC and a PTG.

DEFINITION 7.2. A *semi-PTG* for a language L is a *nondeterministic* Turing machine transducer that on input 1^n outputs some string of length n in L, if such a string exists, and outputs Λ if no such string exists. If G is a semi-PTG, $Gen(G)$ denotes the set of all strings generated by G.

DEFINITION 7.3. A semi-PTG G for a language L is maximal if there exists no semi-PTG G' for L that outputs infinitely more strings than G.

Clearly if a language in NP has a semi-PTG, it has a PTC, and hence it has a PTG, which is a maximal semi-PTG. So the question of whether a language has a maximal semi-PTG is interesting only for languages that are not in NP.

DEFINITION 7.4. A language L is *honestly paddable* if there exists a polynomial-time computable function $P: \Sigma^* \times \Sigma^* \to \Sigma^*$ that is one-to-one in its second argument, such that $x \in L$ if and only if $P(x, y) \in L$ for all x, y, and such that there exists a polynomial p such that $p(|P(x, y)|) \geqq |x| + |y|$ for all x, y.

DEFINITION 7.5. A language L is *augmentable* if there exists a polynomial-time computable function S such that $x \in L$ if and only if $S(x) \in L$, and $|S(x)| > |x|$ for all x. We call S an *augmenting* function for L.

A proof of the following lemma may be found in [20].

LEMMA 7.1. *If a language is honestly paddable, it is augmentable.*

(The proof of the following theorem is a variant of a proof from [20] that shows that recursive sets that are not in P and that are honestly paddable have no maximal P-subsets.)

THEOREM 7.2. *If a language* L *is not in* NP *and is honestly paddable, then it cannot have a maximal semi-*PTG.

Proof. Let L be a language that is not in NP and is honestly paddable. By the lemma, L is augmentable, say with augmenting function S. Suppose G is a maximal semi-PTG for L. Since $Gen(G)$ is in NP, $L - Gen(G)$ is infinite. Let x_0 be any string in $L - Gen(G)$, and consider the sequence $L_{x_0} = \{x_0, x_1, x_2, \cdots, \}$, where $x_i = S(x_{i-1})$ for $i \geqq 1$. Each of these strings is distinct, hence L_{x_0} is an infinite subset of L. Moreover, using G and L_{x_0} we can construct a semi-PTG G_{x_0} that outputs all strings in L_{x_0}, as follows. Given input 1^n, G_{x_0} computes $x_0, x_1, \cdots$ until it finds a string x_k such that $|x_k| \geqq n$. If $|x_k| = n$, G_{x_0} outputs x_k; otherwise, it simulates G on input 1^n and outputs the string produced by G (or Λ if G outputs Λ).

Note that if $L_{x_0} - Gen(G)$ is infinite, then G cannot be maximal and we have derived a contradiction. So we may assume that for each $x_0 \in (L - Gen(G))$ there exists some string $x \in L_{x_0}$ such that $|x| \geqq |x_0|$, $x \notin Gen(G)$, and $S(x) \in Gen(G)$. Consider the set

$$H = \{x \mid x \in L, x \notin Gen(G), \text{ and } S(x) \in Gen(G)\}$$

By the discussion in the above paragraph, each string in $L - Gen(G)$ gives rise to some longer string in H, so H is infinite. Consider the following semi-PTG G_1 for L. On input 1^n G_1 nondeterministically constructs a string y of length n. It then computes $S(y)$ and runs G on input $1^{|S(y)|}$. If G outputs $S(y)$, this means $S(y) \in L \Rightarrow y \in L$, so

G_1 outputs y. Otherwise, G_1 simulates G on input 1^n and outputs the string produced by G. Note that $H \subseteq Gen(G_1)$, so G_1 outputs infinitely more strings than G, which again is a contradiction. Hence G cannot be maximal. □

A type of semi-PTG that is of special interest is a generator that outputs all but a sparse subset of the strings in the language. Clearly if all languages in co-NP have PTGs, then NP = co-NP. This conclusion can also be reached from the weaker assumption given in the following theorem.

THEOREM 7.3. *Suppose each co-*NP *language has a semi-*PTG *that outputs all but a sparse subset of the strings in the language. Then* NP = *co-*NP.

Proof. If all co-NP languages have semi-PTGs, then all co-NP languages have PTCs. By Corollary 6.3, this implies that there are no sparse languages in co-NP − P.

Let L be any language in co-NP, and let G_L be a semi-PTG for L such that $L - Gen(G_L)$ is sparse. Then $L - Gen(G_L)$ is a sparse set in co-NP, and hence in P. Since $Gen(G_L)$ is in NP, L is in NP. Hence NP = co-NP. □

7.3. Variable-length outputs. Another way to make generators less restrictive is to relax the condition that the output string must have exactly the length specified by the input. Since it is desired to output longer strings with longer inputs (otherwise, a trivial generator that outputs only a finite number of strings can always be found), we require that on input 1^n the generator or constructor output a string of length at least n. However, we allow the string to be longer than n, up to some length specified by a function g.

DEFINITION 7.6. A function $g: N^+ \to N^+$ is *unary polynomial-time computable* (*UPTC*) if there exists a polynomial-time Turing machine that given input 1^n outputs $1^{g(n)}$.

DEFINITION 7.7. Let $g: N^+ \to N^+$ *be a* UPTC *function such that* $g(n) \geqq n$ *for all* $n \geqq 1$.

(1) A *g-PTC* for a language L is a *deterministic* Turing machine transducer that runs in polynomial time and that on input 1^n outputs a string in L of length m, for some m such that $n \leqq m \leqq g(n)$, if such a string exists, and outputs Λ otherwise.

(2) A *g-PTG* for a language L is a *nondeterministic* Turing machine transducer that runs in polynomial time and that on input 1^n outputs a string in L of length m, for some m such that $n \leqq m \leqq g(n)$, if such a string exists, and outputs Λ otherwise. Moreover, for each string x in L such that $n \leqq |x| \leqq g(n)$, there exists some computation of the generator on input 1^n that outputs x.

DEFINITION 7.8. A language has a *loose* PTG if it has a g-PTG for some UPTC function g.

As with length-restricted PTGs, if a language has a loose PTG it must be in NP. In addition, for any UPTC function g, an NP language has a g-PTG if and only if it has a g-PTC. If a language has a PTC, it clearly has a g-PTC for any UPTC function g. It would be difficult to exhibit an NP language that has a g-PTC for some function g, but that does not have a length-restricted PTC, since this would imply that not all NP languages have PTCs, and thus that P ≠ NP. The next theorem shows that at least for certain functions g, the existence of g-PTCs implies the existence of length-restricted PTCs.

DEFINITION 7.9. Let L be a language and $g: N^+ \to N^+$ a computable function such that $g(n) \geqq n$ for all $n \geqq 1$.

(1) The function σ_g is defined recursively as follows. $\sigma_g(1) = 1$. For $n > 1$, $\sigma_g(n) = g(\sigma_g(n-1)) + 1$. (Note that $\sigma_g(n) \geqq n$ and that σ_g is a strictly increasing function.)

(2) The language L_g consists of all strings of the form $x*^{\sigma_g(|x|)-|x|}$, where $x \in L$.

THEOREM 7.4. *Suppose g is such that σ_g is unary polynomial-time computable. Then if all* NP *languages have g-*PTCs, *then all* NP *languages have length-restricted* PTCs.

Proof. Suppose all NP languages have g-PTCs and let L be any language in NP. Because σ_g is UPTC, L_g is also in NP, so L_g has a g-PTC C_g. Because $\sigma_g(n+1) > g(\sigma_g(n))$, on input $1^{\sigma_g(n)}C_g$ can only output a string of length exactly $\sigma_g(n)$. So a PTC for L can be defined that on input 1^n simulates C_g on input $1^{\sigma_g(n)}$, and outputs the n-length prefix of the string produced by C_g, if any. □

Which types of function g satisfy the hypothesis of the preceding theorem? The following lemma provides one class of such functions.

LEMMA 7.2. *If g is* UPTC *and σ_g is bounded above by a polynomial, then σ_g is* UPTC.

Proof. This follows from the fact that computation of $\sigma_g(n)$ involves $n-1$ computations of g on arguments, each of which is bounded above by a polynomial in n. □

COROLLARY 7.5. *Let $g(n) = n + c\lceil n^{r/s}\rceil$, where c, r, s ∈ N, $c \geqq 1$, and $0 \leqq r < s$. Then σ_g is* UPTC.

Proof. It can be shown by induction on n that $\sigma_g(n) \leqq c^s n^s$ for $n \geqq 1$. □

(Note that if we set $g(n) = n + cn$, then σ_g is no longer UPTC.)

THEOREM 7.6. *Let L be a language in P. The following are equivalent*:

(1) *L is augmentable.*

(2) *Both L and $\bar{L}$ have loose* PTGs, *and there exists a polynomial p such that for all sufficiently large positive integers n there exist strings $x \in L$, $y \in \bar{L}$ such that $n \leqq |x| \leqq p(n)$ and $n \leqq |y| \leqq p(n)$.*

Proof. (1)⇒(2): Suppose L is augmentable with augmenting function S. Let x_0 be the smallest string in L and let $|x_0| = n_0$. Let p be a nondecreasing polynomial such that $|S(x)| \leqq p(|x|)$ for all x. We define a p-PTC C for L as follows. On input 1^n, if $n < n_0$, C outputs Λ, and if $n = n_0$ C outputs x_0. Otherwise, C computes the sequence of strings $x_1, x_2 \cdots$ where for $i > 0$, $x_i = S(x_{i-1})$, until an x_k is found for which $|x_k| \geqq n$. C then outputs x_k. Note that since $|x_{k-1}| < n$, $|x_k| = |S(x_{k-1})| \leqq p(|x_{k-1}|) \leqq p(n)$. An identical argument shows that $\bar{L}$ also has a p-PTC.

(2)⇒(1): Assume L has a g_1-PTC C_1 and $\bar{L}$ has a g_2-PTC C_2. Let polynomial p have the properties specified in (2). We define an augmenting function S for L as follows. If $x \in L$ and $|x| = n$, determine the smallest $m \geqq n+1$ such that $g_1(m) \geqq p(n+1)$. Note we must have $m \leqq p(n+1)$. Run C_1 on inputs $1^{n+1}, \cdots, 1^m$ until C_1 outputs something other than Λ. This must occur at some point since there exists a string in L having length between $n+1$ and $p(n+1)$. Let $S(x)$ equal the string produced by C_1. Note $|S(x)| \geqq n+1 > |x|$. If $x \in \bar{L}$ compute $S(x)$ similarly using C_2. □

COROLLARY 7.7. *If L is honestly paddable, then L has a loose* PTG.

Proof. If L is honestly paddable, then it must be augmentable. Since the proof that (1)⇒(2) above does not use the fact that L is in P, the result follows. □

Theorem 7.6 applies only to languages in P. However, as is the case for length-restricted PTGs, the existence of loose PTGs for NP languages depends only on the existence of loose PTGs for languages in P.

THEOREM 7.8. *All languages in* P *have loose* PTGs *if and only if all languages in* NP *have loose* PTGs.

Proof. Suppose all languages in P have loose PTGs, and let L be any language in NP. As noted in a previous section, L can be "prefixed" by a language in P. In other words, there exists a language L' in P, a strictly increasing polynomial p such that $p(n) \geqq n$ for all $n \geqq 1$, and a polynomial-time computable onto function $g: L' \to L$, such that $|x| = p(|g(x)|)$ for all $x \in L'$. By assumption, L' has an f-PTC $C_{L'}$, where f is bounded above by a polynomial q. We define an h-PTC C_L for L, where h is computed as follows. Given positive integer n, compute $m = f(p(n))$, find the largest k such that

$p(k) \leqq m$, and set $h(n) = k$. We have $h(n) \leqq f(p(n)) \leqq q(p(n))$, so h is unary polynomial-time computable. C_L on input 1^n will simulate $C_{L'}$ on input $1^{p(n)}$. If $C_{L'}$ outputs Λ, then there are no strings having length between $p(n)$ and $f(p(n))$ in L'. This in turn implies that there are no strings having length between n and $h(n)$ in L, so C_L should output Λ. If $C_{L'}$ outputs a string x, then C_L should output $g(x)$. □

8. Relativizations. This section presents relativization results pertaining to some of the open questions regarding the existence of PTCs and PTGs for various language classes. Details of these constructions, which are only sketched below, may be found in [24] and [23].

DEFINITION 8.1. *Let A be an oracle set.*

(1) A language L has a PTC relative to A if there exists a polynomial-time deterministic oracle Turing machine transducer that on input 1^n and querying oracle A outputs some string $x \in L$ of length n, if such a string exists, and outputs Λ otherwise.

(2) A language L has a PTG relative to A if there exists a polynomial-time nondeterministic oracle Turing machine transducer that on input 1^n and querying oracle A outputs some string $x \in L$ of length n, if such a string exists, and outputs Λ otherwise. Moreover, for each string of length n in L there exists some computation of the generator on input 1^n that outputs x.

8.1. Polynomial-time constructors (PTCs) for NP languages.

THEOREM 8.1. *There exists an oracle A such that there exists a sparse language in $(D^P)^A - P^A$ and such that all languages in NP^A have* PTCs *relative to A.*

Proof (sketch). The construction uses techniques from [3], [18], [7], and [8]. Information about PTCs for NP languages is coded into the oracle; for each NP machine and for each length, the lexicographically first string of that length that is accepted by the machine querying the oracle, together with its prefixes, are coded into the oracle A. At the same time strings are put into the oracle to ensure that the language $L_A = \{w \mid (\exists z \in A$ such that $|z| = 2|w|)$ and $\neg(\exists y$ such that $wy \in A$ and $|w| = |y|\}$, which is clearly in $(D^P)^A$, is not in P^A, and is sparse. □

THEOREM 8.2. *There exists an oracle B and a language $L \in P^B$ such that L does not have a* PTC *relative to B.*

This follows from a simple diagonalization construction.

The last two theorems show that the question of whether or not all NP (or P) languages have PTGs cannot be resolved using techniques that relativize, even under the assumption that $\mathrm{P} \neq \mathrm{NP}$. Note also that Theorem 8.1 shows that the converse of Theorem 4.3 cannot be proved using techniques that relativize.

8.2. Categorical polynomial-time generators (PTGs).

THEOREM 8.3. *There exists an oracle C such that all languages in NP^C have* PTCs *relative to C but such that there exists a language $L \in \mathrm{P}^C$ that does not have a categorical* PTG *relative to C.*

Proof (sketch). As is done in the construction for Theorem 8.1, information is coded into the oracle C to ensure that all languages in NP^C have PTCs relative to C. At the same time strings are put into the oracle to ensure that the language $C \in \mathrm{P}^C$ does not have a categorical PTG relative to C. □

COROLLARY 8.4. *There exists an oracle* D *such that there are no sparse languages in $\mathrm{NP}^D - \mathrm{P}^D$ and such that there exists a language in P^D that does not have a categorical* PTG *relative to* D.

8.3. Polynomial-time constructors (PTCs) for co-NP languages. The following results can be proved using techniques from [3].

THEOREM 8.5. *There exists an oracle E such that $\mathrm{NP}^E = co\text{-}NP^E$ and such that there exists a language in P^E that does not have a* PTC *relative to E.*

THEOREM 8.6. *There exists an oracle F such that $\mathrm{NP}^F \neq co\text{-}\mathrm{NP}^F$ and such that there exists a language in P^F that does not have a* PTC *relative to F.*

THEOREM 8.7. *There exists an oracle G such that all languages in NP^G have* PTCs *relative to G, and such that there exists a language in co-NP^G that does not have a* PTC *relative to G.*

8.4. P-immunity and PTCs (polynomial-time constructors) for NP. Any infinite set that has a PTC has an infinite subset in P, namely, the set of strings that are generated by the PTC. It therefore follows that if all NP languages have PTCs, then NP has no P-immune sets. The converse cannot be proved using techniques that relativize, since combining the result of Theorem 8.2 and an oracle construction by Blum and Impagliazzo [4] relative to which no infinite NP language is P-immune, the following can be derived.

THEOREM 8.8. *There exists an oracle H such that NP^H has no infinite* P*-immune sets but such that there exists a language in P^H that does not have a* PTC *relative to H.*

Acknowledgments. We would like to thank Joel Seiferas for very useful comments and suggestions, and for pointing out that if a language in P has a PTC it has a PTG. We are also grateful to Lane Hemachandra for discussions.

REFERENCES

[1] M. ABADI, E. ALLENDER, A. BRODER, J. FEIGENBAUM, AND L. A. HEMACHANDRA, *On generating solved instances of computational problems*, in Proc. of CRYPTO 88, Santa Barbara, CA, August 1988.

[2] E. BACH, *How to generate factored random numbers*, SIAM J. Comput., 17 (1988), pp. 179–193.

[3] T. BAKER, J. GILL, AND R. SOLOVAY, *Relativizations of the* P = ?NP *question*, SIAM J. Comput., 4 (1975), pp. 431–442.

[4] M. BLUM AND R. IMPAGLIAZZO, *Generic oracles and oracle classes*, in Proc. 28th Annual Symposium on Foundations of Computer Science, Los Angeles, CA, October 1987, pp. 118–126.

[5] R. V. BOOK, *Tally languages and complexity classes*, Inform. and Control, 26 (1974), pp. 186–193.

[6] A. BORODIN AND A. DEMERS, *Some comments on functional self-reducibility and the* NP *hierarchy*, Tech. Report 76-284, Cornell University, Department of Computer Science, Ithaca, NY, July 1976.

[7] J. Y. CAI, T. GUNDERMANN, J. HARTMANIS, L. A. HEMACHANDRA, V. SEWELSON, K. WAGNER, AND G. WECHSUNG, *The boolean hierarchy* I: *Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[8] ———, *The boolean hierarchy* II: *Applications*, SIAM J. Comput., 18 (1989), pp. 95–111.

[9] J. FEIGENBAUM, R. J. LIPTON, AND S. R. MAHANEY, *A completeness theorem for almost-everywhere invulnerable generators*, AT&T Bell Laboratories Tech. Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, February 1989.

[10] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability–A Guide to the Theory of* NP*-Completeness*, W. H. Freeman, New York, NY, 1979.

[11] J. GROLLMANN AND A. L. SELMAN, *Complexity measures for public-key cryptosystems*, SIAM J. Comput., 17 (1988), pp. 309–335.

[12] J. HARTMANIS, V. SEWELSON, AND N. IMMERMAN, *Sparse sets in* NP-P: EXPTIME *versus* NEXPTIME, in Proc. 15th Annual ACM Symposium on Theory of Computing, Boston, MA, April 1983, pp. 382–391.

[13] J. HARTMANIS, *On sparse sets in* NP-P. Inform. Process. Lett., 16 (1983), pp. 55–60.

[14] L. A. HEMACHANDRA, E. ALLENDER, J. FEIGENBAUM, M. ABADI, AND A. BRODER, *Generating hard, certified elements of* NP*-complete sets*, AT&T Bell Laboratories Tech. Memorandum, AT&T Bell Laboratories, Murray Hill, NJ, February 1989.

[15] L. A. HEMACHANDRA, *On ranking*, in Proc. 2nd Annual Conference on Structure in Complexity Theory, June 1987, pp. 103–117.

[16] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to Automata Theory, Languages, and Computation*, Addison-Wesley, Reading, MA, 1979.

[17] M. R. JERRUM, L. G. VALIANT, AND V. V. VAZIRANI, *Random generation of combinatorial structures from a uniform distribution*, Theoret. Comput. Sci., 43 (1986), pp. 169–188.
[18] S. A. KURTZ, *Sparse sets in* NP-P: *Relativizations*, SIAM J. Comput., 14 (1985), pp. 113–119.
[19] I. NIVEN AND H. S. ZUCKERMAN, *An Introduction to the Theory of Numbers*, John Wiley, New York, 1972.
[20] P. ORPONEN, D. A. RUSSO, AND U. SCHÖNING, *Optimal approximations and polynomially levelable sets*, SIAM J. Comput., 15 (1986), pp. 399–408.
[21] C. H. PAPADIMITRIOU AND M. YANNAKAKIS, *The complexity of facets (and some facets of complexity)*, J. Comput. System. Sci., 28 (1984), pp. 244–259.
[22] V. R. PRATT, *Every prime has a succinct certificate*, SIAM J. Comput., 4 (1975), pp. 214–220.
[23] L. A. SANCHIS, *Language Instance Generation and Test Case Construction for* NP-*hard Problems*, Ph.D. thesis, Tech. Report 296, University of Rochester, Computer Science Department, Rochester, NY, May 1989.
[24] L. A. SANCHIS AND M. A. FULK, *Efficient language instance generation*, Tech. Report 235, University of Rochester, Computer Science Department, Rochester, NY, January 1988.
[25] L. J. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1976), pp. 1–22.
[26] G. TINHOFER, *On the generation of random graphs with given properties and known distribution*, Appl. Comput. Sci., 13 (1979), pp. 265–297.
[27] L. G. VALIANT, *Relative complexity of checking and evaluating*, Inform. Process. Lett., 5 (1976), pp. 20–23.
[28] N. C. WORMALD, *Generating random regular graphs*, J. Algorithms, 5 (1984), pp. 247–280.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 297-321, April 1990

RED-BLUE INTERSECTION DETECTION ALGORITHMS, WITH APPLICATIONS TO MOTION PLANNING AND COLLISION DETECTION*

PANKAJ K. AGARWAL† AND MICHA SHARIR‡

Abstract. Let Γ be a collection of n (possibly intersecting) "red" Jordan arcs of some simple shape in the plane and let Γ' be a similar collection of m "blue" arcs. Several efficient algorithms are presented for detecting an intersection between an arc of Γ and an arc of Γ'. (i) If the arcs of Γ' form the boundary of a simply connected region, then the following can be detected: a "red-blue" intersection in time $O(\lambda_s(m)\log^2 m+(\lambda_s(m)+n)\log(n+m))$ where $\lambda_s(m)$ is the (almost-linear) maximum length of (m, s) Davenport-Schinzel sequences, and where s is a fixed parameter, depending on the shape of the given arcs. Another case where an intersection in close to linear time can be detected is when the union of the arcs of Γ and the union of the arcs of Γ' are both connected. (ii) In the most general case, an intersection in time $O((m\sqrt{\lambda_s(n)}+n\sqrt{\lambda_s(m)})\log^{1.5}(m+n))$ can be detected. For several special but useful cases, in which many faces in the arrangements of Γ and Γ' can be computed efficiently, randomized algorithms that are better than the general algorithm are obtained. In particular when all arcs in Γ and Γ' are line segments, a randomized $O((m+n)^{4/3+\varepsilon})$ intersection detection algorithm, for any $\varepsilon>0$, is obtained. The algorithm in (i) is applied to obtain an $O(\lambda_s(n)\log^2 n)$ algorithm (for some small $s>0$) for planning the motion of an n-sided simple polygon around a right-angle corner in a corridor, improving a previous $O(n^2)$ algorithm of [Proc. 4th Annual Symposium on Computational Geometry, Association for Computing Machinery, New York, 1988, pp. 187-192], and to derive an efficient technique for fast collision detection for a simple polygon moving (translating and rotating) in the plane along a prescribed path.

Key words. arrangements, collision detection, complexity of many faces, Davenport-Schinzel sequences, line sweep, motion planning, random sampling, ray-shooting

AMS(MOS) subject classifications. 52A37, 68Q20, 68Q25, 68R99

1. Introduction.

Consider the following problem. Let Γ be a collection of n (possibly intersecting) "red" Jordan arcs in the plane, and let Γ' be a similar collection of m "blue" arcs. Does any red arc intersect any blue arc? Suppose the arcs have relatively simple shape; for example, suppose they are line segments, or x-monotone algebraic arcs of some small fixed degree, so that in particular each pair of them intersects in at most some fixed number of points. There is a simple way to detect a red-blue intersection, using a standard sweep-line algorithm, such as that of Bentley and Ottmann [BO79]. However, this algorithm runs in time $O((m+n+t)\log(m+n))$, where t is the number of red-red and blue-blue intersections that the algorithm has to sweep through before detecting a red-blue intersection. Since t can be quadratic (that is, $\Omega(m^2+n^2)$) in the worst case, the algorithm is not generally efficient, and in fact might probably be inferior to the naive $O(mn)$ algorithm that checks all possible pairs of a red arc and a blue arc for intersection.

* Received by the editors April 11, 1988; accepted for publication (in revised form) April 11, 1989. A preliminary version of this paper has appeared in the Proceedings of the 4th Annual Symposium on Computational Geometry, 1988, pp. 70-80. Work on this paper has been supported by Office of Naval Research grant N00014-82-K-0381, by National Science Foundation grant NSF-DCR-83-20085, and by grants from the Digital Equipment Corporation, and the IBM Corporation. Work by the second author has also been supported by a research grant from the Israeli National Council for Research and Development (NCRD).

† Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012. Present address, DIMACS Center, Rutgers University, Hill Center for the Mathematical Sciences, Busch Campus, P.O. Box 1179, Piscataway, NJ 08859-1179.

‡ Courant Institute of Mathematical Sciences, New York University, 251 Mercer Street, New York, New York 10012, and the School of Mathematical Sciences, Tel-Aviv University, Ramat-Aviv, Tel-Aviv 69978, Israel.

In this paper, we present several efficient algorithms for detecting a red-blue intersection that avoid the overhead of having to examine many red-red or blue-blue intersections. Our algorithms are based on the recently developed theory of Davenport-Schinzel sequences (see [HS86] and [AShSh87]). The maximum length $\lambda_s(m)$ of an (m, s) Davenport-Schinzel sequence is almost linear in m for any fixed value of s, and satisfies the following bounds (where $\alpha(m)$ is the extremely slowly growing functional inverse of Ackermann's function):

$$\lambda_1(m) = m, \quad \lambda_2(m) = 2m - 1 \quad \text{(trivial)},$$

$$\lambda_3(m) = \Theta(m\alpha(m)) \quad [\text{HS86}],$$

$$\lambda_4(m) = \Theta(m \cdot 2^{\alpha(m)}) \quad [\text{AShSh87}],$$

$$\lambda_{2s+2}(m) = O(m \cdot 2^{\alpha^s(m)(1+o(1))}) \quad \text{for } s > 1 \quad [\text{AShSh87}]$$

$$\lambda_{2s+3}(m) = O(m \cdot 2^{\alpha^s(m)\log\alpha(m)(1+o(1))}) \quad \text{for } s \geqq 1 \quad [\text{AShSh87}],$$

$$\lambda_{2s+2}(m) = \Omega(m \cdot 2^{\Omega(\alpha^s(m))}) \quad \text{for } s > 1 \quad [\text{AShSh87}].$$

Our first algorithm considers the case when the red arcs in Γ do not cross one another and form the boundary of a simply connected region, for example, segments bounding a simple polygon. In this case, we can detect a red-blue intersection in time

$$O(\lambda_{s+2}(m)\log^2 m + (\lambda_{s+2}(m) + n)\log(m+n))$$

where s is the maximum number of intersection points between a pair of arcs of Γ'. Thus we obtain an almost linear solution to the intersection detection problem in this special case. In the more special case, where each red arc and each blue arc is a straight-line segment, we can use the ray-shooting technique of [CG89] to detect an intersection of each red segment with the blue polygon in $O(\log n)$ time, after $O(n \log \log n)$ preprocessing, thus obtaining an improved algorithm for intersection detection. However, for general arcs, no such efficient "arc-shooting" procedure is known.

Our algorithm uses a recent result of [GSS89], which shows that the combinatorial complexity of a single face in an arrangement of n arcs, each pair intersecting in at most s points, is $O(\lambda_{s+2}(n))$, and that such a face can be computed in time $O(\lambda_{s+2}(n)\log^2 n)$. We also extend our algorithm to arbitrary collections of intersecting arcs Γ and Γ', provided the union of all red arcs is a connected set, as is the union of all blue arcs; in this extension the algorithm runs in time $O(\lambda_{s+2}(m+n)\log^2(m+n))$.

These results have several applications in motion planning and collision detection for a simple polygon. For example, Maddila and Yap [MY86] study the problem of moving (translating and rotating) an n-sided simple polygon $\mathcal{B}$ around a right-angle corner in a corridor. They show that the problem can be reduced to testing a single canonical motion of $\mathcal{B}$ for intersection with the corridor walls, which in turn can be reduced, by studying the problem in a coordinate frame attached to $\mathcal{B}$, to testing whether $\mathcal{B}$ intersects some resulting collection of algebraic arcs describing the relative motion of the corners and walls of the corridor. Our intersection detection algorithm can then be used to obtain an $O(\lambda_s(n)\log^2 n)$ motion planning algorithm (for some small $s > 0$), considerably improving the $O(n^2)$ algorithm given in [MY86]. More generally, we can use our procedure to obtain fast algorithms for testing a prescribed motion of a simple or "curved" polygon $\mathcal{B}$ for collision with a given collection of obstacles. Under reasonable assumptions on the simplicity of the motion of $\mathcal{B}$ (for example, along algebraic paths of low degree) and on the shape of the boundaries of the obstacles and of $\mathcal{B}$, we can obtain close to linear collision detection procedures.

In the general case of the red-blue intersection detection problem, our algorithms are less efficient, but are still significantly better than quadratic algorithms. In a nutshell, our first algorithm is efficient because it can restrict the problem of detecting a red-blue intersection to a single face in the blue or the red arrangement. For arbitrary red and blue arrangements this is not possible in general, and we need to calculate and search for an intersection in many such faces. We then face the problem of obtaining sharp bounds for the complexity of many such faces, and of their efficient calculation. For general arcs, when such procedures are not available, we present a deterministic algorithm that detects a red-blue intersection in time

$$O((m\sqrt{\lambda_{s+2}(n)}+n\sqrt{\lambda_{s+2}(m)})\log^{1.5}(m+n)).$$

For certain special types of arcs, for which many faces in their arrangements can be computed efficiently, we obtain improved (albeit randomized) algorithms. For example, if all arcs are line segments or unit circles, then we can detect an intersection using a randomized algorithm whose expected running time is $O((m+n)^{4/3+\varepsilon})$, for any $\varepsilon>0$, exploiting the results of [CEGSW88], [EGS88], and [EGH*89].

We believe our algorithms can be extended to obtain all red-blue intersections in an output-sensitive fashion; if all arcs are line segments, then we have an algorithm that produces all k red-blue intersections in time $O((m\sqrt{n}+n\sqrt{m}+k)\log(m+n))$, using the ray-shooting technique of [CG89].

The paper is organized as follows. Section 2 presents the efficient algorithm for the case of a simple (curved) polygon, and § 3 gives applications of this algorithm to motion planning and collision detection. Section 4 describes algorithms for general red-blue intersection detection. Section 5 mentions a few additional applications of the general algorithm. Finally, in § 6 we conclude by mentioning some open problems.

2. Intersection between a simple polygon and arcs. In this section, we give an efficient algorithm for the case when the red arcs in Γ are nonintersecting line segments that form the boundary of a simple polygon $\mathcal{B}$ (see Fig. 1) or, more generally, nonintersecting Jordan arcs whose union is the boundary of a simply-connected region.

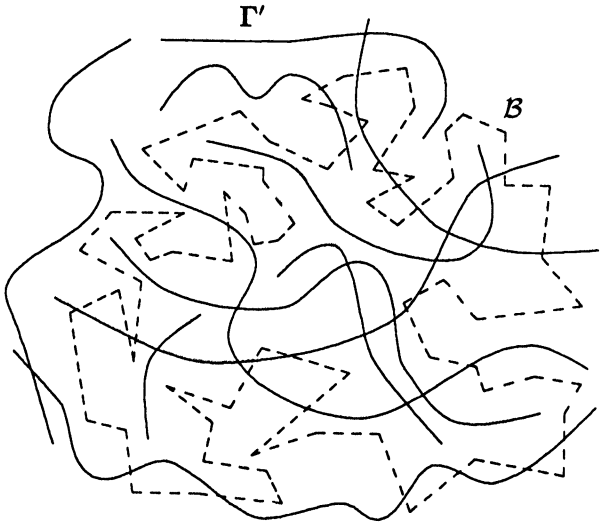


FIG. 1. *Red arcs and blue polygon.*

Let $\mathcal{A}(\Gamma')$ denote the *arrangement* of Γ', that is, the planar map induced by these arcs; its vertices are the intersection points of the arcs and their endpoints, its edges are maximal connected subsets of the arcs that do not contain any vertex, and its faces are maximal connected subsets of the plane that do not contain a vertex or edge. Let z denote a distinguished point on $\partial\mathcal{B}$. The following lemma is a simple but key observation that allows us to obtain an efficient algorithm.

LEMMA 2.1. *Let $\mathcal{F}$ be the face of $\mathcal{A}(\Gamma')$ containing the point z. Then $\partial\mathcal{B}$ and Γ' intersect if and only if $\partial\mathcal{F}$ and $\partial\mathcal{B}$ intersect.*

Proof. The "if" part is obvious. For the "only if" part, simply follow $\partial\mathcal{B}$ from z in, say, the clockwise direction until the first intersection with Γ' is encountered, which necessarily lies on $\partial\mathcal{F}$. □

The above lemma suggests that we do not have to look at the entire arrangement of Γ', whose combinatorial complexity might be $\Omega(m^2)$ in the worst case, but instead we only need to compute a single face of $\mathcal{A}(\Gamma')$. Guibas, Sharir, and Sifrony [GSS88] (see also [PSS88]) have studied the complexity of a single face in an arrangement of Jordan arcs. They have shown the following lemma.

THEOREM 2.2 [GSS89]. *Given a collection $\Gamma' = \{\gamma_1, \gamma_2, \cdots, \gamma_m\}$ of m Jordan arcs in the plane so that each pair of them intersect in at most s points, the complexity of a single face of $\mathcal{A}(\Gamma')$ is bounded by $O(\lambda_{s+2}(m))$ and it can be computed in $O(\lambda_{s+2}(m)\log^2 m)$ time, under an appropriate model of computation.*

Remark 2.3. If the arcs in Γ' are closed or bi-infinite Jordan curves, then the bound on the combinatorial complexity of (respectively, the time to compute) a single face of their arrangement can be replaced by $O(\lambda_s(m))$ (respectively, $O(\lambda_s(m)\log^2 m)$) (see [SS87] and [GSS89]).

The combinatorial bound is obtained by showing that the circular sequence of arcs in the order in which they appear along the boundary of the given face can be written as a Davenport–Schinzel sequence of order $s+2$. The algorithm given by [GSS89] uses a divide-and-conquer approach. It divides Γ' into two subsets $\Gamma'_1 = \{\gamma'_1, \cdots, \gamma'_{m/2}\}$ and $\Gamma'_2 = \{\gamma'_{m/2+1}, \cdots, \gamma'_m\}$, and recursively computes the face $\mathcal{F}_1$ of $\mathcal{A}(\Gamma'_1)$ and the face $\mathcal{F}_2$ of $\mathcal{A}(\Gamma'_2)$ containing the point z. The desired face $\mathcal{F}$ is the connected component of $\mathcal{F}_1 \cap \mathcal{F}_2$ that contains the point z. Using a relatively simple line-sweeping algorithm, a subsequent "red-blue merge" procedure then obtains $\mathcal{F}$ from $\mathcal{F}_1$ and $\mathcal{F}_2$ in $O(\lambda_{s+2}(m)\log m)$ time. Hence, the overall running time is $O(\lambda_{s+2}(m)\log^2 m)$.

Remark 2.4. Here and later we assume a model of computation in which various basic operations involving the given arcs are assumed to require $O(1)$ time. These include finding the intersection of a pair of arcs, or of an arc with a line, testing whether a point lies above or below an arc, and finding the points of vertical tangency along an arc. Thus we are more interested in the combinatorial complexity than in the algebraic complexity of manipulating such arcs, which is an interesting subject in its own right.

Having computed $\mathcal{F}$, all we have to do is test whether or not $\partial\mathcal{F}$ and $\partial\mathcal{B}$ intersect. This is easily done by a variant of the Bentley–Ottmann Algorithm [BO79] in time $O((\lambda_{s+2}(m)+n)\log(m+n))$. Thus, we can conclude Theorem 2.5.

THEOREM 2.5. *If the arcs in Γ form the boundary of a simple ("curved") polygon $\mathcal{B}$, then an intersection between Γ and Γ' can be detected in time*

$$O(\lambda_{s+2}(m)\log^2 m + (\lambda_{s+2}(m)+n)\log(m+n))$$

where s is the maximum number of intersections between a pair of arcs in Γ'.

Proof. The proof is immediate from Theorem 2.2 and the discussion given above. □

Remarks 2.6. (i) If the arcs in Γ' are also line segments, then we can use the ray-shooting technique of [CG89] to detect an intersection of a blue arc with the red polygon in $O(\log n)$ time, after $O(n \log n)$ preprocessing.

(ii) If the arcs in Γ' are closed or bi-infinite Jordan curves, we can replace $s+2$ by s in the preceding results.

The above technique can be extended further to detect an intersection between Γ and Γ' when each of $\mathcal{A}(\Gamma)$ and $\mathcal{A}(\Gamma')$ is a connected planar graph. Let p be an endpoint of an arc in Γ' and let $\mathcal{F}$ be the face of $\mathcal{A}(\Gamma)$ containing the point p. Then we have Lemma 2.7.

LEMMA 2.7. *The arcs in Γ and Γ' intersect if and only if $\partial\mathcal{F}$ and Γ' intersect (see Fig. 2).*

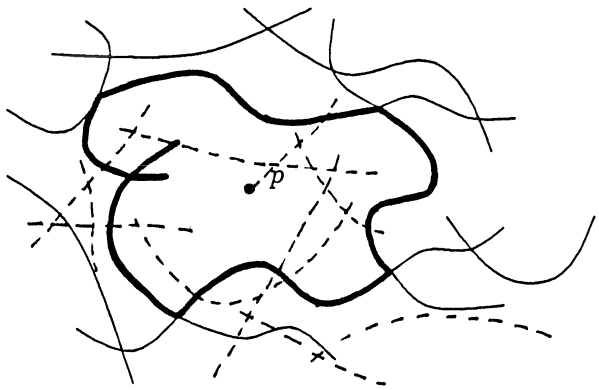


FIG. 2. *$\mathcal{A}(\Gamma)$ and $\mathcal{A}(\Gamma')$ are connected planar graphs: Solid arcs denote Γ, dashed arcs denote Γ', and bold arcs denote $\mathcal{F}$.*

Proof. The "if" part is obvious. For the "only if" part, let q be a red-blue intersection point. Since $\mathcal{A}(\Gamma')$ is a connected graph, there exists a connected path Π from p to q along the edges of $\mathcal{A}(\Gamma')$. Follow Π from p until its first intersection with $\mathcal{A}(\Gamma)$, which necessarily lies on $\partial\mathcal{F}$. □

Thus, we can reduce the problem of detecting an intersection between Γ and Γ' to the problem of detecting an intersection between Γ' and a simply connected region $\mathcal{F}$, which we can solve using the preceding technique. Therefore, we obtain Theorem 2.8.

THEOREM 2.8. *If each of $\mathcal{A}(\Gamma)$ and $\mathcal{A}(\Gamma')$ is a connected planar graph, then an intersection between Γ and Γ' can be detected in time $O(\lambda_{s+2}(m+n)\log^2(m+n))$, where s is the maximum number of intersections between a pair of arcs in Γ or in Γ'.*

Proof. Using Theorem 2.2, we can compute the face $\mathcal{F}$ in time $O(\lambda_{s+2}(n)\log^2 n)$. From Lemma 2.7 it follows that we only have to detect an intersection between $\partial\mathcal{F}$ and Γ' that, by Theorem 2.5 and the following remark, can be done in time

$$O(\lambda_{s+2}(m)\log^2 m + (\lambda_{s+2}(m)+\lambda_{s+2}(n))\log(m+n)),$$

and hence the overall running time is

$$O(\lambda_{s+2}(m)\log^2(m+n)+\lambda_{s+2}(n)\log^2(m+n)) = O(\lambda_{s+2}(m+n)\log^2(m+n)).$$ □

Remarks 2.9. (i) Note that we do not require any bound on the number of intersections between a given blue arc and a given red arc, because our algorithms stop as soon as one such intersection is detected. However, the algorithm still assumes that an intersection between a blue arc and a red arc can be detected in $O(1)$ time.

(ii) If the maximum number of intersection points between a pair of arcs in Γ (respectively Γ') is s (respectively, s'), then the running time of the above algorithm is

$$O((\lambda_{s+2}(n)+\lambda_{s'+2}(m)) \cdot \log^2 (m+n)).$$

Moreover, if the arcs in Γ (respectively, Γ') are closed or bi-infinite curves, then we can replace $s+2$ (respectively $s'+2$) by s (respectively, s').

3. Applications to collision detection and motion planning. Using the intersection detection algorithm of the previous section, we present an efficient algorithm for the following problem:

> Given a simple polygon $\mathcal{B}$ that is allowed to translate and rotate in the plane, a set of polygonal obstacles, and a prescribed continuous motion of $\mathcal{B}$, check whether $\mathcal{B}$ collides with any obstacle during this motion.

We begin by reducing this problem to that studied in the previous section. Let $\mathbf{O}=\{O_1, O_2, \cdots, O_l\}$ denote the given set of polygonal obstacles having pairwise disjoint interiors. We can represent their boundaries as a collection of "walls" $\mathbf{W}=\{W_1, \cdots, W_m\}$ where each W_i is an edge of some obstacle. We also define two coordinate systems: the *environment-frame* and the *object-frame.* The environment-frame is the usual coordinate system in which the obstacles are fixed and $\mathcal{B}$ moves. On the other hand, the object-frame is rigidly attached to $\mathcal{B}$, so in it $\mathcal{B}$ is fixed and the positions of the obstacles vary depending on the position of $\mathcal{B}$ in the environment-frame. The position of $\mathcal{B}$, in the environment-frame, can be described by a pair $Z=[X, \theta]$, where $X=(x, y)$ is the position of a fixed point p in $\mathcal{B}$, which is taken to be the origin of the object-frame, and θ is the orientation of the x-axis p_x of the object-frame, relative to the environment-frame (see Fig. 4 of § 3.2). Formally, for any such placement Z of $\mathcal{B}$ we obtain an Euclidean transformation Ψ_Z that maps a point ξ in the object-frame to its position ω in the environment-frame as follows:

$$x(\omega)=x+x(\xi) \cdot \cos \theta - y(\xi) \cdot \sin \theta, \tag{3.1}$$

$$y(\omega)=y+x(\xi) \cdot \sin \theta + y(\xi) \cdot \cos \theta. \tag{3.2}$$

In the object-frame, the position of any obstacle O_i depends on the position Z of $\mathcal{B}$ in the environment-frame and is obtained by the corresponding inverse map Ψ_Z^{-1}.

Let $\mathbf{\Pi}$ denote a continuous motion of $\mathcal{B}$, given as a continuous path $\mathbf{\Pi}:[0, 1] \to \mathcal{E}^2 \times \mathcal{S}^1$. We call $\mathbf{\Pi}$ a *basic* path if $\mathbf{\Pi}([0, 1])$ is an algebraic curve in x, y and $\tan \theta/2$ having a fixed and small degree. Examples of basic paths are translation along a straight line or along a low-degree algebraic curve, rotation, sliding with a fixed pair of vertices of $\mathcal{B}$ touching a fixed pair of walls, etc. For exposition sake we keep the notion of a basic path somewhat loose and informal and therefore, for example, we do not specify precisely how low do we want its degree to be, etc. We assume that $\mathbf{\Pi}$ is either a basic path or a concatenation of basic paths. The complexity of $\mathbf{\Pi}$, denoted by $|\mathbf{\Pi}|=k$, is the number of basic paths from which it is composed, and we write $\mathbf{\Pi}=\pi_1 \| \pi_2 \| \cdots \| \pi_k$. We use the term $\Psi_{\mathbf{\Pi}}$ to denote the map that sends each point ξ in the object-frame to

$$\{\Psi_{\mathbf{\Pi}(t)}(\xi) \mid t \in [0, 1]\}.$$

$\Psi_{\mathbf{\Pi}}^{-1}$ is defined symmetrically for the inverse transformation Ψ^{-1}.

LEMMA 3.1. *Let q be a fixed point in the environment-frame and let $\mathcal{B}$ move along a basic path $\mathbf{\Pi}$. Then $\Psi_{\mathbf{\Pi}}^{-1}(q)$ is also a basic path (albeit its maximum degree might be higher). Similarly, if W is a wall, then $\Psi_{\mathbf{\Pi}}^{-1}(W)$ is a region whose boundary consists of a constant number of algebraic curves of fixed degree.*

Proof. Let $q=(\rho, \sigma)$ be a fixed point in the environment-frame and let $\Pi(t)=[x(t), y(t), \theta(t)]$ denote the placement of $\mathcal{B}$ at time t as it moves long Π. Suppose $(\xi, \zeta)=\Psi_{\Pi(t)}^{-1}(q)$; then by (3.1) and (3.2)

$$\rho = x(t)+\xi\cdot\cos\theta(t)-\zeta\cdot\sin\theta(t),$$

$$\sigma = y(t)+\xi\cdot\sin\theta(t)+\zeta\cdot\cos\theta(t).$$

Therefore,

$$\xi=(\rho-x(t))\cdot\cos\theta(t)+(\sigma-y(t))\cdot\sin\theta(t), \tag{3.3}$$

$$\zeta=-(\rho-x(t))\cdot\sin\theta(t)+(\sigma-y(t))\cdot\cos\theta(t). \tag{3.4}$$

Since Π is an algebraic curve of fixed degree, $x(t), y(t)$, $\sin\theta(t)$ and $\cos\theta(t)$ (or equivalently, $\tan\theta(t)/2$) are related by a pair of polynomial equations:

$$P\left(x, y, \tan\frac{\theta}{2}\right)=0, \tag{3.5}$$

$$Q\left(x, y, \tan\frac{\theta}{2}\right)=0 \tag{3.6}$$

of some fixed degree. Eliminating these three variables from the four equations (3.3), (3.4), (3.5), and (3.6) (cf. [VdW70]), we get an algebraic curve (in ξ and ζ) of fixed degree, showing that $\Psi_{\Pi}^{-1}(q)$ is a basic path.

Now let us consider the area swept by a wall W (see Fig. 3). Without loss of generality let us assume that W is a portion of the y-axis. It is easily checked that any point on the boundary of $\Psi_{\Pi}^{-1}(W)$ either lies on one of the paths $\Psi_{\Pi}^{-1}(q_1)$, $\Psi_{\Pi}^{-1}(q_2)$, where q_1, q_2 are the endpoints of W, or lies on the *envelope* of the parameteric family of curves $\{\Psi_{\Pi(t)}^{-1}(\ell): t\in[0,1]\}$, where ℓ is the y-axis (see [Co36] for details). For each $t\in[0,1]$, let $\Pi(t)=[x(t), y(t), \theta(t)]$. Then, the points $(\xi, \zeta)\in\Psi_{\Pi(t)}^{-1}(\ell)$ in the object frame satisfy the following equation, as is implied by (3.1), (3.2)

$$x(t)+\xi\cos\theta(t)-\zeta\sin\theta(t)=0. \tag{3.7}$$

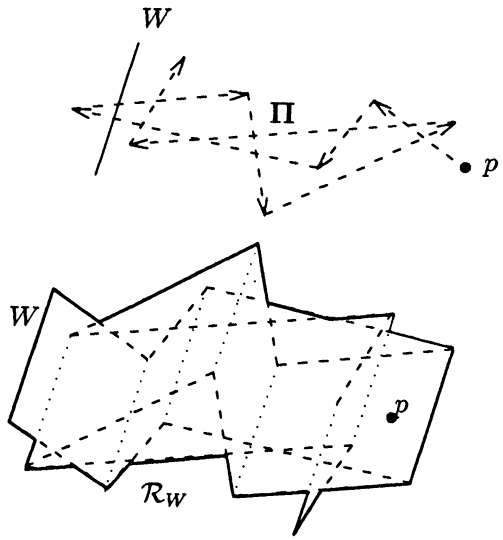


FIG. 3. *Area $\mathcal{R}_W$ swept by a wall W.*

Let $f(\xi, \zeta, t)$ denote the left-hand side of (3.7). The envelope Φ of $f(\xi, \zeta, t) = 0$ (with ξ, ζ as independent variables and t as a parameter) is found by eliminating t from the pair of equations $f(\xi, \zeta, t) = (\partial/\partial t) f(\xi, \zeta, t) = 0$ (see [Co36]). In our case these equations are

$$\xi \cos \theta(t) - \zeta \sin \theta(t) = -x(t),$$

$$\xi \sin \theta(t) + \zeta \cos \theta(t) = \frac{\dot{x}(t)}{\dot{\theta}(t)}.$$

Rewriting these equations, we obtain

$$\begin{aligned} \xi &= -x(t) \cos \theta(t) + \frac{\dot{x}(t)}{\dot{\theta}(t)} \sin \theta(t), \\ \zeta &= x(t) \sin \theta(t) + \frac{\dot{x}(t)}{\dot{\theta}(t)} \cos \theta(t) \end{aligned} \tag{3.8}$$

where both right-hand sides are algebraic in $x(t), y(t), \tan \theta/2$, as is easily checked. Eliminating these three variables as was done in the case of a point shows that the envelope Φ is algebraic (of degree generally larger than that of $\mathbf{\Pi}$, but nevertheless having fixed maximum value, depending on the degree of $\mathbf{\Pi}$). Thus, the boundary of $\Psi_{\mathbf{\Pi}}^{-1}(W)$ consists of a constant number of connected subarcs of three basic paths $\Psi_{\mathbf{\Pi}}^{-1}(q_1), \Psi_{\mathbf{\Pi}}^{-1}(q_2), \Phi$, where this constant depends on the pattern of intersections of these paths. □

3.1. The algorithm. In this section we give an efficient algorithm for the collision detection problem, defined in the beginning of the section. A simple approach to solve this problem is to compute the region swept by every edge e_i of $\mathcal{B}$ as $\mathcal{B}$ moves along $\mathbf{\Pi}$ and test whether it intersects any wall W. Since for each basic path π_j, the region swept by e_i is of fixed complexity, we can check in $O(m)$ time if it intersects with any W. There are k basic paths and n edges, so the total time spent will be $O(k \cdot m \cdot n)$. Using the algorithm of § 2, we show that we can do much better than this naive approach.

Without loss of generality, we can assume that the initial position of $\mathcal{B}$ in $\mathbf{\Pi}$ is a free position. In practical applications this can be expected to be the case; in general, we can run a standard line sweeping algorithm (such as that of [BO79]), to verify that $\mathcal{B}$ starts at a free placement, in $O((n+m) \log (n+m))$ time. Let $\mathcal{R}_j = \mathcal{R}_j(W)$ denote the region in the object-frame swept by the wall W when $\mathcal{B}$ moves along π_j, and let γ_j denote its boundary, that is $\gamma_j = \partial \Psi_{\pi_j}^{-1}(W)$, and let $\mathbf{\Gamma}_W = \{\gamma_1, \gamma_2, \cdots, \gamma_k\}$; by Lemma 3.1, each γ_j is the union of $O(1)$ algebraic arcs.

LEMMA 3.2. *$\mathcal{B}$ intersects a wall W while moving along $\mathbf{\Pi}$ if and only if $\partial\mathcal{B}$ intersects $\Psi_{\mathbf{\Pi}}^{-1}(W)$ in the object frame. Therefore either $\partial\mathcal{B}$ intersects an arc in $\mathbf{\Gamma}_W$, or $\partial\mathcal{B}$ lies "inside" one of the regions $\mathcal{R}_j$.*

Proof. We only prove the first statement of the lemma because the second statement is an obvious consequence. Since the initial position of $\mathcal{B}$ is free, collision between W and $\mathcal{B}$ implies that there exists a placement $Z_0 \in \mathbf{\Pi}$ such that $\partial\mathcal{B}$ and W intersect at some point $\zeta \in \Psi_{Z_0}(\partial\mathcal{B}) \cap W$. But then it follows by definition that

$$\Psi_{Z_0}^{-1}(\zeta) \in \partial\mathcal{B} \cap \Psi_{Z_0}^{-1}(W).$$

Hence, if $\mathcal{B}$ intersects a wall W while moving along $\mathbf{\Pi}$, then $\partial\mathcal{B}$ intersects the area swept by W in the object frame. □

We thus first test, in $O(mk)$ time, whether the origin p lies in any of the regions $\mathcal{R}_j$. If so, an intersection has been detected and we can stop right away. If not, the

previous lemma implies that it suffices to determine whether $\partial\mathcal{B}$ intersects an arc in $\mathbf{\Gamma} = \bigcup_{W \in \mathbf{W}} \mathbf{\Gamma}_W$. Hence we obtain Theorem 3.3.

THEOREM 3.3. *Given a polygon $\mathcal{B}$ with n vertices, a path $\mathbf{\Pi} = \pi_1 \| \pi_2 \| \cdots \| \pi_k$, composed of k basic paths $\pi_1, \cdots, \pi_k$ and a set of polygonal obstacles $\mathbf{O}$ with a total of m walls, we can check in time*

$$O(\lambda_s(mk) \log^2 (mk) + (n + \lambda_s(mk)) \log (n + mk))$$

if $\mathcal{B}$ collides with any of the obstacles in $\mathbf{O}$ when it moves along $\mathbf{\Pi}$. Here s is a constant depending on the maximum algebraic degree of the basic paths of $\mathbf{\Pi}$.

Proof. It requires only $O(mk)$ time to check whether the origin p lies inside any of the regions $\mathcal{R}_j(W)$, the area swept by the wall W when $\mathcal{B}$ moves along the basic path π_j. We can compute $\mathbf{\Gamma}$, the collection of the boundary arcs of all the regions $\mathcal{R}_j$, in time $O(mk)$, and Theorem 2.5 implies that we can detect an intersection between $\mathbf{\Gamma}$ and $\partial\mathcal{B}$ in time

$$O(\lambda_s(mk) \log^2 (mk) + (n + \lambda_s(mk)) \log (n + mk))$$

where $s-2$ is the maximum number of intersections between a pair of arcs in $\mathbf{\Gamma}$. □

3.2. Applications to motion planning. In this section we describe some applications of the above collision-detection algorithm. This algorithm is useful when $mk \approx n$, because in this case it takes only $O(\lambda_s(n) \log^2 n)$ time to determine if the polygon $\mathcal{B}$ collides with any obstacle in $\mathbf{O}$ while moving along the prescribed path $\mathbf{\Pi}$. Two such typical cases are the following: (i) there are $O(n)$ walls in the environment and the path of $\mathcal{B}$ consists of $O(1)$ basic paths; and (ii) the environment is of a small fixed size, i.e., has only $O(1)$ walls, but the path $\mathbf{\Pi}$ may consist of as many as $O(n)$ basic paths.

On the other hand, if $mk \gg n$, $nk \approx m$, and the number of objects $l = O(1)$ (that is, there are not many obstacles though each obstacle can have many edges), then we can use a similar approach in the environment-frame instead of the object-frame. The basic idea is as follows. Let $\mathcal{R}_j$, for $1 \leqq j \leqq k$, denote the area swept by the edge e of $\mathcal{B}$, when $\mathcal{B}$ moves along the basic path π_j. Let $\gamma_j = \partial\mathcal{R}_j$ and $\mathbf{\Gamma}_e = \{\gamma_1, \cdots, \gamma_k\}$. Using Lemma 3.2 we can again show that $\mathcal{B}$ collides with an obstacle O_r if and only if either O_r lies inside any of $\mathcal{R}_j$ or ∂O_r intersects $\mathbf{\Gamma} = \bigcup_{e \in \mathcal{B}} \mathbf{\Gamma}_e$. Let m_r be the number of edges in O_r. Since it takes only $O(nk)$ time to check whether O_r lies inside any of the regions $\mathcal{R}_j$, it follows from Theorem 2.5 that we can detect an intersection between $\mathbf{\Gamma}$ and ∂O_r in time

$$O(\lambda_{s+2}(nk) \log^2 nk + (m_r + \lambda_{s+2}(nk)) \log (nk + m_r))$$

where s is the maximum number of intersections between any pair of arcs in $\mathbf{\Gamma}$. Summing over all l obstacles, the total time required to detect a collision is

$$O(\lambda_{s+2}(nk) \log^2 nk + (m + \lambda_{s+2}(nk)) \log (nk + m)) = O(\lambda_{s+2}(m) \log^2 m),$$

because $nk \approx m$ and $l = O(1)$. Note that this argument assumes that each obstacle O_r is simply connected.

The main advantage of either of these two variants of our approach is therefore that the resulting complexity bound does not involve the term $O(mn)$, which seems to be unavoidable in any standard motion planning algorithm that explicitly constructs the configuration space.

Returning to our application, there are certain motion planning problems in which we can define a single canonical motion, and show that if there exists a collision-free motion from the initial to the final position, then this canonical path is also collision-free. In such cases we can apply our collision detection technique to obtain an improved

motion planning algorithm. Such a case is given by Maddila and Yap [MY86], who have studied the problem of moving a simple n-sided polygon $\mathcal{B}$ around a right-angle corner in a corridor. They have presented an $O(n^2)$ algorithm for planning such a motion; using our techniques, we obtain an improved, almost linear algorithm.

A corridor with one right-angle corner is an infinite L-shaped region, where the two sides of the corridor extend in the $-x$ direction and the $+y$ direction, respectively (as shown in Fig. 4). The problem of moving a polygon $\mathcal{B}$ around the corner is to move $\mathcal{B}$ from a given initial position $Z_I = [x_I, y_I, \theta_I]$ in the left side of the corridor, to a final position $Z_F = [x_F, y_F, \theta_F]$ in the other side of the corridor, without colliding with the corridor walls. In some applications Z_F may not be given and we only have to find out whether $\mathcal{B}$ can be moved around the corner from some given initial placement Z_I. We will first analyze the former problem and then show how to handle the latter variant.

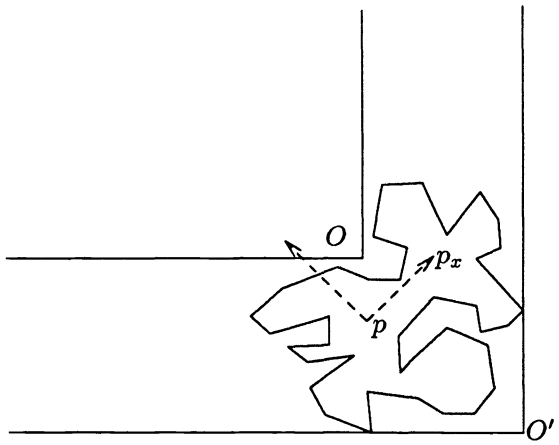


FIG. 4. *Simple polygon inside a corridor.*

It is shown in [MY86] that a collision-free motion from Z_I to Z_F exists if and only if the following canonical motion of $\mathcal{B}$ is collision-free:

(i) Translate in the $-y$-direction, until $\mathcal{B}$ touches the horizontal wall of the convex corner of the corridor.

(ii) Translate in the x-direction until $\mathcal{B}$ touches the vertical wall of the convex corner.

(iii) Slide $\mathcal{B}$ while touching the two walls of the convex corner until the orientation becomes θ_F; this is the only nontrivial part of the motion, and we denote it by Π.

(iv) Translate in the y-direction until $y(p) = y_F$.

(v) Translate in the $-x$-direction until $x(p) = x_F$.

If Z_I and Z_F are free positions, then all of the translational motions are collision-free. Thus all we have to do is calculate the path Π in (iii) and verify that it is also collision-free. This is done by applying our algorithm in the object frame to detect an intersection between $\mathcal{B}$ and the concave corner and adjacent walls of the corridor. The efficiency of the algorithm obviously depends on the complexity of Π, that is, the number of basic paths composing it that is analyzed as follows.

We can parameterize Π using the orientation of $\mathcal{B}$ as it changes monotonically and continuously along Π (see [MY86] for details). At any orientation, the rightmost vertex u of $\mathcal{B}$ touches the vertical wall of the convex corner and the bottommost vertex

v of $\mathcal{B}$ touches the horizontal wall of the corner. Except at a finite number of *critical orientations*, where an edge of the convex hull of $\mathcal{B}$ adjacent to u (respectively, v) becomes vertical (respectively, horizontal), the *supporting pair* (u, v) is unique and does not change between any pair of consecutive critical orientations. The path formed by sliding $\mathcal{B}$ with a fixed pair of vertices touching a fixed pair of walls is known as a *glissette* [Lo61]; each point on $\mathcal{B}$ traces an elliptic arc, and the path (as a mapping into $\mathcal{E}^2 \times \mathcal{S}^1$) is easily seen to be a basic path.

LEMMA 3.4 [MY86]. *The sequence (u_i, v_i) of supporting pairs encountered along Π changes in an orderly manner; (u_i, v_i) change in the order in which they appear along the convex hull of $\mathcal{B}$. Moreover, the number of supporting pairs is at most $2n$ and they can be found in $O(n)$ time.*

The above lemma implies that Π consists of at most $2n$ basic paths, and by Lemma 3.1 the arcs describing the "inverse" motion of the walls in the object frame of $\mathcal{B}$ are algebraic curves of some fixed low degree. Thus, we obtain Theorem 3.5.

THEOREM 3.5. *It takes $O(\lambda_s(n) \log^2 n)$ time (for some small value of $s > 0$) to determine whether $\mathcal{B}$ can be moved around a right-angle corner in a corridor.*

Proof. The proof is immediate from Lemma 3.4 and Theorem 3.3, noting that m is constant and $k \leqq 2n$. □

If Z_F is not specified, the moving $\mathcal{B}$ around the corner amounts to sliding it as in the above canonical motion until it reaches an orientation θ_F at which it can be translated vertically into the upper portion of the corridor, for which it is necessary and sufficient that the width of $\mathcal{B}$ in the direction perpendicular to θ_F should be less than the width of the upper portion of the corridor. This observation enables us to find the smallest such θ_F in $O(n)$ time using the standard "rotating calipers" method. Having found θ_F, the problem reduces to the one just studied, and the same algorithm can be applied.

We can extend our algorithm to a more general corridor in which the exterior angle of the concave corner is not less than the interior angle of the convex corner (see Fig. 5(a)). Indeed, our algorithm relies on the ability to reduce any collision-free motion of $\mathcal{B}$ around the corner to a canonical one in which $\mathcal{B}$ slides along the walls of the lower convex portion of the corridor. The conditions assumed are easily seen to imply that a "retraction" similar to that used above, that is first translate $\mathcal{B}$ to the right until it touches the right wall, and then slide it downward along the walls until the lowest vertex of $\mathcal{B}$ touches the lower horizontal wall of the corridor, can be applied in this case. Once again, we can show that this canonical motion consists of at most $2n$ basic paths, leading to an algorithm similar to that presented above. Figure 5(b)

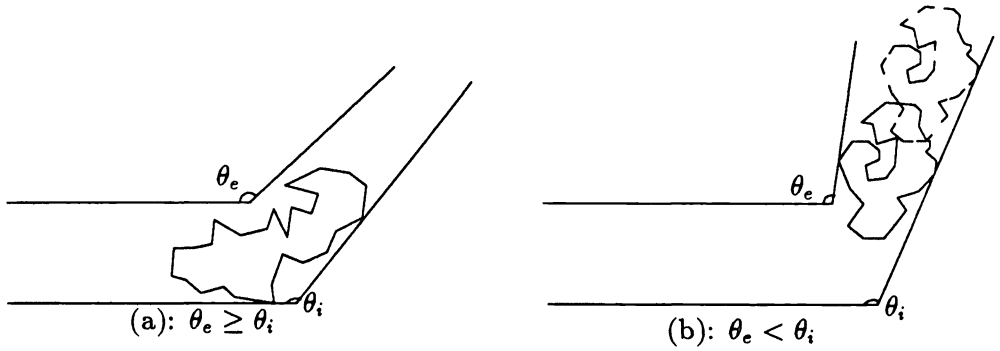


FIG. 5. *General corridors.*

gives an example where such a retraction is not continuous when the above conditions are not satisfied.

If $\mathcal{B}$ is a star-shaped polygon, we can somewhat improve the algorithm, because we can replace the calculation of a single face in $\mathcal{A}(\mathbf{\Gamma})$ (as in [GSS89]) by the calculation of a certain lower envelope of $\mathbf{\Gamma}$, where $\mathbf{\Gamma}$ is the collection of arcs describing the inverse motions of the walls as above. To this end, let ζ be a point in the kernel of $\mathcal{B}$, and consider each arc γ of $\mathbf{\Gamma}$ as a function $r=\gamma(\theta)$ in polar coordinates about ζ in the object frame; if necessary, we break each arc γ into a constant number of pieces, each monotone with respect to θ. The lower envelope $\mathcal{M}_{\mathbf{\Gamma}}$ of $\mathbf{\Gamma}$ in these coordinates is defined as

$$\mathcal{M}_{\mathbf{\Gamma}}(\theta)=\min_{\gamma\in\mathbf{\Gamma}}\{\gamma(\theta)\}.$$

LEMMA 3.6. *$\mathcal{B}$ collides with a wall forming the concave corner of the corridor while moving along the canonical path if and only if $\mathcal{B}$ intersects the lower envelope $\mathcal{M}_{\mathbf{\Gamma}}$ defined above.*

Proof. Lemma 3.2 implies that $\mathcal{B}$ collides with a wall W while moving along the canonical path $\mathbf{\Pi}$ if and only if $\mathcal{B}$ and $\Psi_{\mathbf{\Pi}}^{-1}(W)$ intersect. Therefore, intersection of $\mathcal{M}_{\mathbf{\Gamma}}$ and $\mathcal{B}$ implies that $\mathcal{B}$ collides with some W while moving along $\mathbf{\Pi}$.

Conversely, let ξ be an intersection point of $\mathcal{B}$ and $\partial\Psi_{\mathbf{\Pi}}^{-1}(W)$. The point ξ lies on one of the curves in $\mathbf{\Gamma}$, and if $\xi\notin\mathcal{M}_{\mathbf{\Gamma}}$, then some other point σ on the segment $\overline{\zeta\xi}$ must lie on this lower envelope. Since $\mathcal{B}$ is star-shaped and ζ lies in its kernel, the entire segment $\overline{\zeta\xi}$, and thus also σ, lies inside $\mathcal{B}$. Therefore, $\mathcal{M}_{\mathbf{\Gamma}}$ and $\mathcal{B}$ intersect. □

An intersection between $\mathcal{M}_{\mathbf{\Gamma}}$ and $\mathcal{B}$ can be easily detected as follows. Let a *breakpoint* of $\mathcal{M}_{\mathbf{\Gamma}}$ denote either an endpoint of some arc $\gamma\in\mathbf{\Gamma}$, a point of radial tangency on such an arc, or a point where $\mathcal{M}_{\mathbf{\Gamma}}$ is simultaneously attained by two arcs in $\mathbf{\Gamma}$. Sort the breakpoints of $\mathcal{M}_{\mathbf{\Gamma}}$ in angular order about ζ and similarly sort the vertices of $\mathcal{B}$ in angular order about ζ. Note that the latter sorted list is the same as the order of the vertices of $\mathcal{B}$ along its boundary. Merge these two lists to obtain an overall list $\mathcal{L}$ of orientations about ζ, with the property that for each interval $\mathcal{I}$ between any pair of consecutive orientations in $\mathcal{L}$, $\mathcal{M}_{\mathbf{\Gamma}}$ is attained over $\mathcal{I}$ by a single arc $\gamma_{\mathcal{I}}$ and the portion of $\partial\mathcal{B}$ seen from ζ in directions in $\mathcal{I}$ is a portion of a single edge $e_{\mathcal{I}}$. We can thus detect in constant time an intersection between $\mathcal{B}$ and $\mathcal{M}_{\mathbf{\Gamma}}$ within the angular sector defined by $\mathcal{I}$. There are $O(\lambda_s(n))$ breakpoints in $\mathcal{M}_{\mathbf{\Gamma}}$ (for some small fixed $s>0$), so the length of $\mathcal{L}$ and the time required to detect an intersection between $\mathcal{M}_{\mathbf{\Gamma}}$ and $\mathcal{B}$ are both $O(\lambda_s(n))$. $\mathcal{M}_{\mathbf{\Gamma}}$ can be calculated (in the required sorted order) in time $O(\lambda_s(n)\log n)$, using a standard divide-and-conquer technique [At85]. We thus have Theorem 3.7.

THEOREM 3.7. *It takes only $O(\lambda_s(n)\log n)$ time (for some small $s>0$) to determine whether a star-shaped polygon $\mathcal{B}$ can be moved around a corner in a corridor.*

Remark 3.8. A similar technique applies if $\mathcal{B}$ is a monotone polygon, in which case the radial lower envelope is replaced by appropriate lower and upper envelopes in the direction of monotonicity of $\mathcal{B}$; we leave details of this easy extension to the reader.

4. Red-blue intersection detection in general. If the arcs in $\mathbf{\Gamma}$ or in $\mathbf{\Gamma}'$ do not form a simply connected region, or they are arbitrarily intersecting and their arrangements are disconnected, then Lemma 2.1 does not hold and we may have to search for an intersection in more than one face of $\mathcal{A}(\mathbf{\Gamma}')$. Two difficulties can then arise. One is that the complexity of many faces in such an arrangement can be much higher than

linear. Second, the algorithm of [GSS89] does not generalize to many faces, at least not for general arcs.

As discussed earlier, a sweep-line approach is doomed to be quadratic in the worst case, because it may have to sweep across many red-red or blue-blue intersections. A technique of Chazelle for point location in algebraic manifolds [Ch85] (see also [CS88]) can be used in the case of algebraic arcs to obtain a slightly subquadratic algorithm, but otherwise we are not aware of any previous algorithm that achieves substantially subquadratic performance for this general red-blue intersection detection problem. In this section we present algorithms of this kind. We first describe a deterministic algorithm that works for general arcs, and then obtain in several restricted but useful special cases more efficient randomized algorithms.

Throughout this section we assume that the arcs in Γ and Γ' are x-monotone. Note that if the arcs are algebraic curves, or intersect a vertical line in at most $O(1)$ points, this assumption does not pose any restriction because any nonmonotone arc can be split into $O(1)$ x-monotone subarcs. Note that this assumption is also implicit in the algorithm of [GSS89], which we have used in § 3.

4.1. Deterministic algorithm. Our deterministic algorithm consists of three phases. In the first phase, we partition Γ' into $\ell' < m/r'$ sets $\Gamma'_0, \Gamma'_1, \Gamma'_2, \cdots, \Gamma'_{\ell'}$, where r' is a parameter to be chosen later, so that (i) for $1 \leqq j \leqq \ell'$, every Γ'_j has $r' \leqq m_j < 2r'$ arcs and $\mathcal{A}(\Gamma'_j)$ forms a connected planar graph, (ii) Γ'_0 (possibly empty) contains the remaining arcs of Γ' and each connected component[1] of $\mathcal{A}(\Gamma'_0)$ has less than r arcs. For each $1 \leqq j \leqq \ell'$, we check whether an arc of Γ'_j intersects an arc of Γ, using a modified version of the technique of § 2 (to be described below). In the second phase, we partition Γ into $\ell < n/r$ sets, $\Gamma_0, \Gamma_1, \Gamma_2, \cdots, \Gamma_\ell$, satisfying conditions analogous to those of the partition of Γ', and for each $1 \leqq j \leqq \ell$, we check whether Γ_j intersects Γ'. Finally, in the third phase we check whether Γ'_0 and Γ_0 intersect. Together, the three phases will detect an intersection between Γ and Γ', if one exists.

The first phase of the algorithm proceeds as follows. We partition Γ' into $\Gamma'_0, \Gamma'_1, \cdots, \Gamma'_{\ell'}$ using a line sweep approach [SH76]. We sweep Γ' from left to right with a vertical line; every time we sweep through an intersection point of two arcs, we merge the connected components of $\mathcal{A}(\Gamma')$ containing them, and whenever a component is found to contain more than r' arcs, we remove all these arcs from Γ'. In more detail, as in any standard line sweep algorithm we store the arcs currently intersecting the sweep line in a list Q, sorted in increasing order of y coordinate of their intersections with the sweep line. In addition we maintain the connected components of $\mathcal{A}(\overline{\Gamma'} \cap h^-)$, where $\overline{\Gamma'}$ is the set of arcs that have been encountered by the sweep line so far but have not yet been deleted by the algorithm, and h^- is the half plane lying to the left of the sweep line. For each arc $\gamma' \in \Gamma'$, we store the connected component $\mathcal{C}(\gamma')$ that contains it. If an arc γ' has not been encountered by the sweep line, $\mathcal{C}(\gamma')$ is not defined. At every event point of the sweep, that is an endpoint of one arc or an intersection point between two arcs in $\overline{\Gamma'}$, we perform the standard list and priority queue updating operations as well as the following additional tasks.

(i) If we reach the left endpoint of some arc γ', then we create a new connected component $\mathcal{C}(\gamma')$ containing only γ'. Sweeping through the right endpoint of an arc requires no special action.

[1] In what follows we use the term "connected component" of an arrangement to refer to a component of the union of its arcs, whereas the term "face" will continue to refer to a connected component of the complement of the union of these arcs.

(ii) If we reach an intersection point of two arcs γ' and γ'', and $\mathscr{C}(\gamma')$ is different from $\mathscr{C}(\gamma'')$, then we merge the two connected components containing γ' and γ'', respectively.

(iii) If any connected component $\mathscr{C}$ is found to have more than r' arcs, then we output it as a set Γ'_j, and delete $\mathscr{C}$ and all of its arcs from $\overline{\Gamma'}$ as well as Q. Note that $\mathscr{C}$ need not be a full component of $\mathscr{A}(\Gamma')$, but may be only a subset of such a component. We also remove all endpoints and intersection points along these arcs from the priority queue. The priority queue is also updated by inserting into it all the intersection points between newly adjacent pairs of arcs in Q.

It is easy to see that the number of arcs in each connected component processed during the sweep is between r' and $2r'$. The arcs not deleted by the sweep are put into the set Γ'_0. See below for analysis of the time complexity of this sweep.

Next for each $1 \leqq j \leqq \ell'$, we check whether Γ'_j intersects Γ using the following procedure.

We first construct the arrangement $\mathscr{A}(\Gamma'_j)$ of the arcs in Γ'_j using a line sweep technique. If the two endpoints of an arc $\gamma \in \Gamma$ lie in two different faces of $\mathscr{A}(\Gamma'_j)$, then obviously γ intersects some arc of Γ'_j. If they lie in the same face f_i of $\mathscr{A}(\Gamma'_j)$, then γ need not intersect Γ'_j; however if it does, it has to intersect a subarc lying on the boundary of f_i (as in Lemma 2.1). Therefore the next step is to determine, for each arc $\gamma \in \Gamma$, the faces that contain the endpoints of γ. This can be easily done during the line sweeping procedure that produces $\mathscr{A}(\Gamma'_j)$ (see also [Pr79]), in overall time $O((r'^2+n)\log(r'+n))$. To reiterate, our sweep performs the following steps for each $\gamma \in \Gamma$.

(i) If the two endpoints of γ lie in two different faces of $\mathscr{A}(\Gamma'_j)$, then we have found a red-blue intersection and we stop.

(ii) If the two endpoints of γ lie in the same face f_i of $\mathscr{A}(\Gamma'_j)$, we assign γ to f_i. Thus if the sweep does not detect an intersection, it produces a partition $\Gamma_{f_1}, \cdots, \Gamma_{f_p}$ of Γ, for $p = O(r'^2)$, such that all arcs in Γ_{f_i} have both of their endpoints in the face f_i of the arrangement. Let $n_i = |\Gamma_{f_i}|$. As argued above, it suffices to check the arcs of each Γ_{f_i} for intersection with the edges of f_i. A simple way of doing it is to take an arc of Γ_{f_i} and check its intersection with all edges of f_i. But in the worst case f_i may be bounded by $\Omega(r')$ arcs and all arcs of Γ may lie in this face, in which case this naive procedure will be too expensive. The following lemma suggests a way to improve this procedure by exploiting the property that the boundary of each face in $\mathscr{A}(\Gamma'_j)$ is connected.

LEMMA 4.1. *Let f_i be a simply connected face of $\mathscr{A}(\Gamma'_j)$, and let $\mathscr{F}_i$ be the unbounded face of $\mathscr{A}(\Gamma_{f_i})$. An arc $\gamma \in \Gamma_{f_i}$ intersects Γ'_j if and only if $\partial\mathscr{F}_i$ and ∂f_i intersect (see Fig. 6). For unbounded faces f_i of $\mathscr{A}(\Gamma'_j)$, take $\mathscr{F}_i$ to be the face of $\mathscr{A}(\Gamma_{f_i})$ that contains some point of ∂f_i.*

Proof. Using the argument of Lemma 2.1, we can prove that Γ_{f_i} and Γ'_j intersect if and only if ∂f_i and Γ_{f_i} intersect. By assumption, ∂f_i is a simple closed Jordan curve, or a simple unbounded Jordan arc. Take any point $z \in \partial f_i$, and let $\mathscr{F}_i$ be the face of $\mathscr{A}(\Gamma_{f_i})$ that contains z; it is easily verified, using the x-monotonicity of the arcs in Γ_{f_i}, that if f_i is bounded then $\mathscr{F}_i$ is the unbounded face of $\mathscr{A}(\Gamma_{f_i})$. The claim now follows from another application of Lemma 2.1, in which the roles of Γ and Γ' are interchanged. □

We thus compute one face $\mathscr{F}_i$ for each Γ_{f_i}, as prescribed by Lemma 4.1, and then detect an intersection between any ∂f_i and $\partial\mathscr{F}_i$ by performing another line sweep.

This completes the description of the first phase of our algorithm. The second phase proceeds in a completely symmetric manner, where we partition Γ into connected

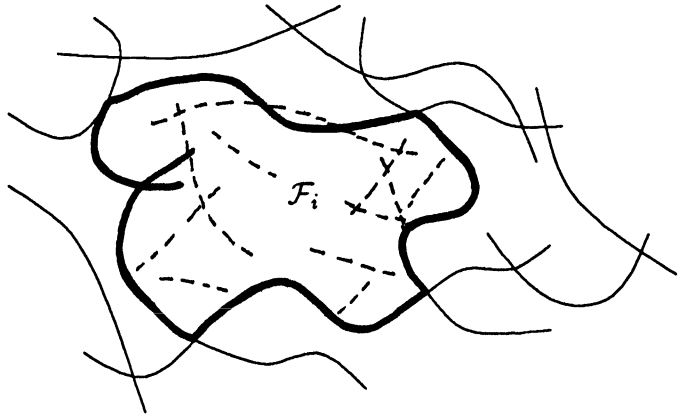


FIG. 6. *Illustration for Lemma* 4.1.

components of size roughly r (to be specified later) instead of r'. For the final phase, we first construct the full arrangements $\mathcal{A}(\mathbf{\Gamma}'_0)$ and $\mathcal{A}(\mathbf{\Gamma}_0)$. Then by sweeping the two arrangements with a vertical line we check whether an edge of $\mathcal{A}(\mathbf{\Gamma}'_0)$ intersects an edge of $\mathcal{A}(\mathbf{\Gamma}_0)$.

The correctness of the algorithm follows easily from the analysis given above. As to analysis of its time complexity, we first need the following simple lemma.

LEMMA 4.2. *If each connected component of* $\mathcal{A}(\mathbf{\Gamma})$ *has at most* r *arcs, then* $\mathcal{A}(\mathbf{\Gamma})$ *has only* $O(nr)$ *vertices.*

Proof. Let $\mathbf{\Gamma}_1, \cdots, \mathbf{\Gamma}_k$ denote the sets of arcs constituting each connected component of $\mathcal{A}(\mathbf{\Gamma})$. If two sets $\mathbf{\Gamma}_i$ and $\mathbf{\Gamma}_j$ have at most $r/2$ arcs each, we merge them. We repeat this process until there are no two such subsets. Then each $\mathbf{\Gamma}_i$ (except possibly one) has at least $r/2$ but less than r arcs, so the number of these sets is at most $2n/r+1$ and each $\mathcal{A}(\mathbf{\Gamma}_i)$ has $O(r^2)$ vertices. Since $\mathcal{A}(\mathbf{\Gamma}_i)$ and $\mathcal{A}(\mathbf{\Gamma}_j)$, for $j \neq i$, do not intersect, there are only $(2n/r+1) \times O(r^2) = O(nr)$ vertices in $\mathcal{A}(\mathbf{\Gamma})$. □

LEMMA 4.3. *Partitioning* $\mathbf{\Gamma}'$ *into* $\mathbf{\Gamma}'_1, \cdots, \mathbf{\Gamma}'_{\ell'}$ *in the first phase of the algorithm can be accomplished in* $O(mr' \log m)$ *time.*

Proof. Inspecting the line sweeping algorithm used to produce the partitioning, we see that most of its steps can be performed quite efficiently. For example, maintaining the connected components of $\mathcal{A}(\overline{\mathbf{\Gamma}'})$ (creating new connected components and merging pairs of components) can be easily done in time $O(m\alpha(m))$ using the union-find data structure [Ta83]. Also, the time needed to delete arcs from Q every time a component is produced is $O(m \log m)$ because each arc is deleted only once. The most expensive part of the algorithm is the maintenance of the priority queue, which takes time $O(K \log m)$, where K is the number of endpoints and intersection points encountered by the algorithm. Therefore, to prove the lemma, we only have to show that $K = O(mr')$.

Let $\mathbf{\Gamma}'_0, \mathbf{\Gamma}'_1, \cdots, \mathbf{\Gamma}'_{\ell'}$ be the connected components produced by the algorithm. Since the algorithm merges connected components whenever it sweeps through an intersection point of two arcs lying in two different components, it never encounters an intersection point between arcs of two different $\mathbf{\Gamma}'_i$ and $\mathbf{\Gamma}'_j$; more precisely, it may have added such points into the priority queue, but it never reaches these points during the sweep. Moreover, at every intersection swept by the line, only a constant number of events are added to the priority queue. Hence, the total number K of intersections encountered by the algorithm is proportional to the number of intersections swept through, which in turn is at most the total number of vertices in all the subarrangements

for $\mathscr{A}(\mathbf{\Gamma}'_j)$ for $0 \leqq j \leqq \ell'$. For $j \geqq 1$, $\mathbf{\Gamma}'_j$ has less than $2r'$ arcs, therefore $\mathscr{A}(\mathbf{\Gamma}'_j)$ has $O(r'^2)$ vertices. Furthermore, each connected component of $\mathscr{A}(\mathbf{\Gamma}'_0)$ has less than r' arcs, so by Lemma 4.2 $\mathscr{A}(\mathbf{\Gamma}'_0)$ has only $O(mr')$ vertices. Since $\ell' < \lceil m/r' \rceil$, the total number of intersection points K is $O(mr')$. Hence, it takes only $O(mr' \log m)$ time to partition $\mathbf{\Gamma}'$ into $\mathbf{\Gamma}'_0, \cdots, \mathbf{\Gamma}'_{\ell'}$. □

LEMMA 4.4. *The time spent in the first phase of the algorithm is bounded by*

$$O\left(mr' \log (m+n) + \frac{m}{r'} \lambda_{s+2}(n) \log^2 n\right).$$

Proof. It follows from the previous lemma that $\mathbf{\Gamma}'_0, \cdots, \mathbf{\Gamma}'_{\ell'}$ can be obtained in time $O(mr' \log m)$. Therefore, we only have to bound the time spent in detecting an intersection between $\mathbf{\Gamma}'_j$, for $1 \leqq j \leqq \ell'$, and $\mathbf{\Gamma}$.

If $|\mathbf{\Gamma}_{f_i}| = n_i$, then Theorem 2.2 implies that $\partial \mathscr{F}_i$ can be computed in time $O(\lambda_{s+2}(n_i) \log^2 n_i)$. By using a line sweep technique we can determine in time

$$O((m_i + \lambda_{s+2}(n_i)) \log (n_i + m_i))$$

whether $\partial \mathscr{F}_i$ and f_i intersect, where m_i is the number of edges in ∂f_i. Summing these costs over all faces f_i of $\mathscr{A}(\mathbf{\Gamma}'_j)$, we get

$$\sum_{i=1}^{p} O(m_i \log (n_i + m_i) + \lambda_{s+2}(n_i) \log^2 n_i)$$

$$= O\left(\sum_{i=1}^{p} m_i \log (n_i + m_i) + \sum_{i=1}^{p} \lambda_{s+2}(n_i) \log^2 n_i\right).$$

The total number of edges in $\mathscr{A}(\mathbf{\Gamma}'_j)$ is bounded by $O(r'^2)$, so $\sum_{i=1}^{p} m_i = O(r'^2)$. Since every arc of $\mathbf{\Gamma}$ is in exactly one $\mathbf{\Gamma}_{f_i}$, $\sum_{i=1}^{p} n_i = n$, and $\mathscr{A}(\mathbf{\Gamma}'_j)$ and $\mathbf{\Gamma}_{f_1}, \cdots, \mathbf{\Gamma}_{f_p}$ can be computed in time $O(r'^2 \log r')$ and $O((r'^2 + n) \log (r' + n))$, respectively. Thus, the total time spent in detecting an intersection between $\mathbf{\Gamma}$ and $\mathbf{\Gamma}'_j$ is

$$O(r'^2 \log (r' + n) + \lambda_{s+2}(n) \log^2 n).$$

Since, $l' < \lceil m/r' \rceil$, the overall time spent in the first phase, including the overhead of partitioning $\mathbf{\Gamma}$, is at most

$$O(mr' \log m) + O\left(\frac{m}{r'} r'^2 \cdot \log (n + r') + \frac{m}{r'} \lambda_{s+2}(n) \cdot \log^2 n\right)$$

$$= O\left(m \cdot r' \log (n+m) + \frac{m}{r'} \cdot \lambda_{s+2}(n) \log^2 n\right). \qquad \square$$

Using a symmetric analysis, it follows that the second phase requires

$$O\left(n \cdot r \log (m+n) + \frac{n}{r} \cdot \lambda_{s+2}(m) \log^2 m\right)$$

time. As to the third phase, Lemma 4.2 implies that $\mathscr{A}(\mathbf{\Gamma}'_0)$ has only $O(mr')$ vertices therefore, we can easily compute it in time $O(mr' \log m)$. Similarly, $\mathscr{A}(\mathbf{\Gamma}_0)$ has $O(nr)$ vertices and can be computed in time $O(nr \log n)$. Once we have $\mathscr{A}(\mathbf{\Gamma}'_0)$ and $\mathscr{A}(\mathbf{\Gamma}_0)$, we can easily check in time $O((mr' + nr) \log (m+n))$ whether they intersect, which bounds the running time of the third phase. We thus obtain the main result of this section.

THEOREM 4.5. *Given a set of n "red" Jordan arcs Γ, and another set of m "blue" Jordan arcs Γ', we can detect in time*

$$O((m\sqrt{\lambda_{s+2}(n)}+n\sqrt{\lambda_{s+2}(m)})\log^{1.5}(m+n))$$

whether Γ and Γ' intersect, where s is the maximum number of intersections between a pair of arcs in Γ or in Γ'.

Proof. Summing the cost of all the three phases of the algorithm and choosing $r'=(\lambda_{s+2}(n)\log(m+n))^{1/2}$, $r=(\lambda_{s+2}(m)\log(m+n))^{1/2}$, the overall running time becomes

$$O((m\sqrt{\lambda_{s+2}(n)}+n\sqrt{\lambda_{s+2}(m)})\log^{1.5}(m+n)).$$ □

Remarks 4.6. In some special cases we can do somewhat better as follows.

(i) If the maximum number of intersections between a pair of arcs in Γ (respectively, Γ') is s (respectively, s'), then the running time of the above algorithm becomes

$$O((m\sqrt{\lambda_{s+2}(n)}+n\sqrt{\lambda_{s'+2}(m)})\log^{1.5}(m+n)).$$

(ii) If Γ' can be partitioned into subsets of the appropriate size r so that the boundary of each face in the arrangement of each subset of Γ is simply connected, then we do not need the last two phases of the above algorithm. Therefore, it easily follows that the running time of the algorithm becomes $O(m\sqrt{\lambda_{s+2}(n)}\log^{1.5} n)$, which is certainly better than the running time of the above algorithm for $n \gg m$. A case in which we can ensure that each f_i is simply connected is when the arcs in Γ' are concatenated to one another at their endpoints to form a single connected (possibly self-intersecting) chain. This situation typically arises in collision-detection problems of the sort studied in §§ 2 and 3.

(iii) Another special case in which we can get a slightly improved bound is when the arcs in Γ are nonintersecting. Again, we do not need the last two phases. We partition Γ' arbitrarily into $\lceil m/r' \rceil$ set Γ'_j, each of size at most r', and then , for each Γ'_j, as in the first phase of the general algorithm, we compute $\Gamma_{f_1}, \cdots, \Gamma_{f_p}$. Since arcs in Γ are nonintersecting we do not have to compute any specific face of Γ_{f_i}, but instead we can detect an intersection between ∂f_i and Γ_{f_i} by performing a line sweep. By choosing $r'=\sqrt{n}$, we can easily check that the overall running time is bounded by $O(m\sqrt{n}\log n)$.

(iv) Again, if all the arcs in Γ (respectively, Γ') are closed or bi-infinite Jordan curves, then we can replace $s+2$ (respectively, $s'+2$) by s (respectively, s') in the preceding results.

4.2. Reporting red-blue intersections. In this section we consider the problem of reporting all red-blue intersections. If the arcs in Γ and in Γ' are nonintersecting, Mairson and Stolfi [MS88] have given an optimal $O((m+n)\log(m+n)+K)$ algorithm to report all K red-blue intersections. In the special case where all arcs are line segments, the algorithm by Chazelle and Edelsbrunner [CE88] for segment intersections can also be applied to report all red-blue intersections in the same time (see also [Cl89] and [Mu88]); unlike [MS88], the algorithm of [CE88] cannot be extended to general arcs. However, when red-red or blue-blue intersections exist, all these algorithms are forced to process them as well, thus failing to achieve output-sensitive behavior. Here $s>0$.

We also do not have an algorithm for the general case, but if all the arcs in Γ and Γ' are (possibly intersecting) line segments, we can extend our previous algorithm to report all red-blue intersections in time that depends on the output size. Again, the algorithm consists of three phases. We partition Γ into subsets Γ_j, for $0 \leqq j \leqq \ell$, and

Γ' into subsets Γ'_j, for $0 \leqq j \leqq \ell'$, as described above. In the first phase we report the intersections between Γ and $\Gamma' - \Gamma'_0$, in the second phase we report the intersections between Γ'_0 and $\Gamma - \Gamma_0$, and finally in the third phase we report the intersections between Γ'_0 and Γ_0.

The first phase proceeds as follows. As in the detection algorithm, we consider each Γ'_j separately. Each face of $\mathscr{A}(\Gamma'_j)$ is a simple polygon, or "semisimple" in the sense that its boundary may touch itself when two sides of a segment bound the same face, but cannot cross itself. We preprocess each face of $\mathscr{A}(\Gamma'_j)$ for answering ray-shooting queries using the techniques of [CG89] or [GHLST87] (which also apply to semisimple polygons). Let ρ and σ be the endpoints of a segment $\gamma \in \Gamma$ and let f_ρ, f_σ be the faces of $\mathscr{A}(\Gamma'_j)$ containing these endpoints, as determined in the first phase of the detection algorithm. By performing a ray-shooting query we find out the edge e of f_ρ that $\overrightarrow{\rho\sigma}$ hits first. If there is no such edge, then γ does not have any red-blue intersection; otherwise the intersection point η is obviously a red-blue intersection and we report it. Let f_η be the other face of $\mathscr{A}(\Gamma'_j)$ incident to e. We now repeat the same process for the segment $\overrightarrow{\eta\sigma}$ in f_η, We continue this until we reach the other endpoint σ of γ. Since for each arc $\gamma \in \Gamma$ all ray-shooting queries except the last one return distinct red-blue intersections, we can charge the last query to γ and the remaining queries to the intersections. The preprocessing of face f_i requires $O(|f_i| \log |f_i|)$ time, or $O(|f_i| \log \log |f_i|)$ using a more involved algorithm of [GHLST87], and a ray-shooting query can be answered in time $O(\log r')$. Therefore all k'_j red-blue intersections between Γ and Γ'_j can be reported in time $O(n \log r' + r'^2 \log r' + k'_j \log r')$. Using the same argument as in Lemma 4.4 we can show that the total time spent in the first phase is $O(mr' \log (m+n) + (nm/r') \log m + k' \log m)$, where k' is the number of red-blue intersections between Γ and $\Gamma' - \Gamma'_0$. Similarly, all k red-blue intersections between Γ'_0 and $\Gamma - \Gamma_0$ can be reported in the second phase in time $O(nr \log (n+m) + (mn/r) \log n + k \log n)$. Since in the third phase we can afford the calculation of red-red and blue-blue intersections, we can report all red-blue intersections by sweeping over all the arcs in Γ'_0 and Γ_0. Thus, summing up all the costs and choosing $r' = \sqrt{n}$ and $r = \sqrt{m}$, we obtain Theorem 4.7.

THEOREM 4.7. *If all the arcs in Γ and Γ' are line segments, then all K red-blue intersections can be reported in time $O((m\sqrt{n} + n\sqrt{m} + K) \log (m+n))$.*

Remark 4.8. The above algorithm also shows that if all arcs are line segments, then the red-blue intersection detection problem can be solved deterministically in time $O((m\sqrt{n} + n\sqrt{m}) \log (m+n))$, which is slightly better than the running time of the general algorithm. See the concluding section for recent developments that imply an improvement of this bound (using a considerably more complicated algorithm).

4.3. Relation between many faces and red-blue intersection detection. In this section we explore the close relationship between the problem of computing many faces in the arrangement of a given set of Jordan arcs and the red-blue intersection detection problem for those arcs. That is, we show that an efficient algorithm for the former problem can be used to obtain an efficient algorithm for the latter one. The basic idea is that a red-blue intersection can be detected by knowing only a few faces of $\mathscr{A}(\Gamma)$ and $\mathscr{A}(\Gamma')$. We formalize this idea in the following lemma.

LEMMA 4.9. *Let $\mathscr{F} = \{\partial f_1, \partial f_2, \cdots, \partial f_k\}$ (respectively, $\mathscr{F}' = \{\partial f'_1, \partial f'_2, \cdots, \partial f'_{k'}\}$) be the collection of the (connected components of the) boundaries of the faces in $\mathscr{A}(\Gamma)$ (respectively, $\mathscr{A}(\Gamma')$) that are either unbounded or contain at least one endpoint of a bounded arc either in Γ or Γ'. Then Γ and Γ' intersect if and only if $\mathscr{F}$ and $\mathscr{F}'$ intersect.*

Proof. The "if" part is obvious. For the "only if" part, suppose first that all arcs in Γ and Γ' are bounded, and again recall our assumption that all these arcs are x-monotone. Let σ be a red-blue intersection point. Let $\mathcal{A}(\Gamma_\sigma)$ (respectively, $\mathcal{A}(\Gamma'_\sigma)$) denote the connected component of $\mathcal{A}(\Gamma)$ (respectively, $\mathcal{A}(\Gamma')$) containing the point σ (see Fig. 7). It is obvious that the rightmost point r' of $\mathcal{A}(\Gamma'_\sigma)$ lies in its unbounded face f', therefore its boundary $\partial f'$ is in the set $\mathcal{F}'$. Let r be the rightmost endpoint of $\mathcal{A}(\Gamma_\sigma)$; again, r must lie in the unbounded face f of $\mathcal{A}(\Gamma_\sigma)$, so that ∂f is in $\mathcal{F}$. Without loss of generality assume that r is to the right of r', which implies that r lies in f'.

There exists a connected path Π in $\mathcal{A}(\Gamma_\sigma)$ from r to σ. Follow $\partial f'$ from r' until a red-blue intersection point σ^* is encountered. Such an intersection must be encountered because either σ lies on $\partial f'$ or $\partial f'$ intersects Π. Since $\sigma^* \in \partial f'$ and also on the boundary of the face of $\mathcal{A}(\Gamma)$ containing r', $\mathcal{F}$ and $\mathcal{F}'$ intersect.

If Γ or Γ' contains unbounded arcs, we reduce the analysis to the bounded arcs by drawing a sufficiently large circle that contains all bounded arcs and intersection points of $\Gamma \cup \Gamma'$, cuts each unbounded arc in two points, and traverses only unbounded faces of $\mathcal{A}(\Gamma)$, $\mathcal{A}(\Gamma')$. The "clipped" arrangement now consists of only bounded arcs, and the unbounded face of each of them is the union of all unbounded faces in the original arrangement. Applying the preceding analysis, the claim follows easily in this case too. □

If $\Gamma \cup \Gamma'$ has t unbounded arcs, then there are at most $2(m+n-t)$ endpoints and the total number of unbounded faces is at most $2t$. Therefore, the above lemma shows that it is enough to compute only $2(m+n-t)+2t = 2(m+n)$ faces of each of $\mathcal{A}(\Gamma)$ and $\mathcal{A}(\Gamma')$ instead of computing the whole arrangement that can have $\Omega(n^2)$ (or $\Omega(m^2)$) faces. Let $\beta(k, n)$ (respectively, $\beta'(k, m)$) denote the maximum number of edges in k distinct faces of $\mathcal{A}(\Gamma)$ (respectively, $\mathcal{A}(\Gamma')$), and let $\Lambda(k, n)$ (respectively, $\Lambda'(k, m)$) denote the time required to compute these faces. We can easily show Theorem 4.10.

THEOREM 4.10. *Let Γ be a collection of n red Jordan arcs and let Γ' be a collection of m blue Jordan arcs. Then, in the above notation, an intersection between Γ and Γ' can be detected in time*

$$O(\Lambda(2(m+n), n)+\Lambda'(2(m+n), m) + (\beta(2(m+n), n)+\beta'(2(m+n), m)) \cdot \log(m+n)).$$

Proof. By the previous lemma, it suffices to compute $2(m+n)$ faces of $\mathcal{A}(\Gamma)$ and of $\mathcal{A}(\Gamma')$, which can be done in time $\Lambda(2(m+n), n)$ and $\Lambda(2(m+n), m)$, respectively. Once having computed these faces, an intersection between them can be detected in

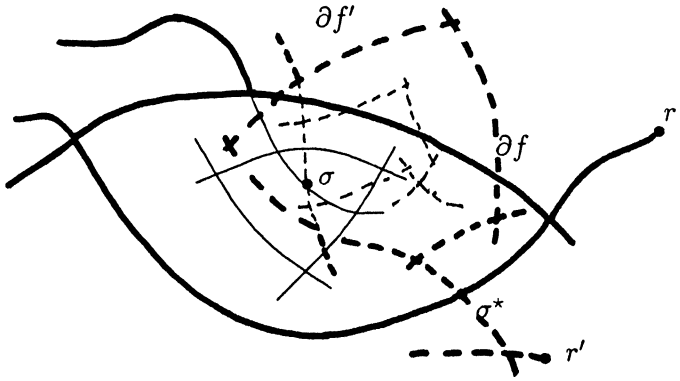


FIG. 7. *Illustration for Lemma* 4.9.

time $O((\beta(2(m+n), n)+\beta'(2(m+n), m))\log(m+n))$ by using a line sweep. Thus, the overall running time is bounded by

$$O(\Lambda(2(m+n), n)+\Lambda'(2(m+n), m) + (\beta(2(m+n), n)+\beta'(2(m+n), m))\cdot\log(m+n)).$$ □

[CEGSW88], [EGS88], and [EGH*89] have studied the complexity of many faces in arrangements of certain classes of Jordan curves and for some special cases they have obtained bounds that are either optimal or optimal within logarithmic factors, together with comparably efficient randomized algorithms for calculating many such faces, which are based on the random sampling technique of [Cl87] and [HW87]. Using these results and Theorem 4.10, we obtain Corollary 4.11.

COROLLARY 4.11. *If each of Γ and Γ' is a set of line segments, of unit circles, or of pseudolines (that is, unbounded arcs, each pair of which intersect at most once), then an intersection between Γ and Γ' can be detected by a randomized algorithm whose expected running time is $O((m+n)^{4/3+\varepsilon})$, for any $\varepsilon>0$ (where the constant of proportionality depends on ε). If Γ or Γ' is a set of arbitrary circles, and the other set is either a set of curves in one of the above classes or is also a set of arbitrary circles, then an intersection between Γ and Γ' can be detected in randomized expected time $O((m+n)^{7/5+\varepsilon})$, for any $\varepsilon>0$.*

Proof. The proof follows immediately from the results of [EGS88], [CEGSW88], and [GSS89]. For line segments, in [EGS88] it has been shown that m faces in an arrangement of n line segments have $O(m^{2/3-\varepsilon}n^{2/3+2\varepsilon}+n\alpha(n))$ edges, and that they can be computed in randomized expected time

$$O(m^{2/3-\varepsilon}n^{2/3+2\varepsilon}+n\alpha(n)\log^2 n\log m),$$

for any $\varepsilon>0$. In this case we have

$$\begin{aligned}\Lambda(m+n, n) &= \Lambda'(m+n, m)\\ &= (\beta(2(m+n), n)+\beta'(2(m+n), m))\log(m+n)\\ &= O((m+n)^{4/3+\varepsilon})\end{aligned}$$

for any $\varepsilon>0$, so Theorem 4.10 implies the claim. Similar bounds for arrangements of the other kind of curves, as established in [CEGSW88] and [GSS89], imply the claim in all other cases. □

If the arcs in Γ are nonintersecting, then $\mathcal{A}(\Gamma)$ has only one face and there are only n edges in it. Moreover, it is enough to detect an intersection between Γ and the faces of $\mathcal{A}(\Gamma')$ that are unbounded or contain the endpoints of bounded arcs in Γ. There are at most $2n$ faces of $\mathcal{A}(\Gamma')$ containing endpoints of arcs in Γ; the unbounded faces of $\mathcal{A}(\Gamma')$ can be regarded as a single connected face if we clip each unbounded arc of Γ' at two sufficiently distant points, as in the proof of Lemma 4.9. We thus obtain Corollary 4.12.

COROLLARY 4.12. *If the arcs in Γ are nonintersecting, then an intersection between Γ and Γ' can be detected in time $O(\Lambda'(2n, m)+\beta'(2n, m)\log(m+n))$, where $\Lambda'(n, m)$ is the time needed to compute n faces of $\mathcal{A}(\Gamma')$, and $\beta'(n, m)$ is the maximum number of edges in n such faces.*

5. More applications. In this section, we mention some applications of the general red-blue intersection detection algorithms. We describe only two applications and leave it to the imagination of the reader to roam free in search of further applications.

Consider the following problem:

> Given a billiard table in the shape of an arbitrary k-sided convex polygon, and n billiard balls of equal size lying on the table, suppose we shoot one of these balls B in a given direction, and allow it to bounce m times off the sides of the table. Will B hit any of the other balls?

This problem can be solved using our red-blue intersection detection algorithm as follows. Since all balls have the same radius, it is easy to see that if B collides with any other ball B_i, it does so on the plane that passes through the centers of all the balls. Without loss of generality let us assume that this is the xy-plane. Therefore we can consider this problem as a two-dimensional collision-detection problem, in which the moving object is a circle, all obstacles are also circles, and we want to determine whether the moving circle collides with any other circle while following the path of B.

Let γ_i^* denote the circle obtained by projecting B_i on the xy-plane and let γ^* be the projection of B. We expand each circle γ_i^* by the radius of γ^*, so we obtain a set of n possibly intersecting circles, Γ (see Fig. 8). Next we shrink the table by the radius of B, let R denote this convex polygon; R has at most k edges. It is easy to see that this procedure reduces the moving ball to a single point z in the sense that B collides with any other ball B_i if and only if z intersects any circle of Γ while moving on the path followed by the center of B (strictly speaking, the projection of the path on the xy-plane). The path Π followed by z can be determined in time $O(m \log k)$ after $O(k)$ preprocessing, using a trivial ray-shooting algorithm in a convex polygon. Let Γ' denote the set of segments forming Π. It follows from the above discussion that B collides with any other ball B_i if and only if Γ and Γ' intersect.

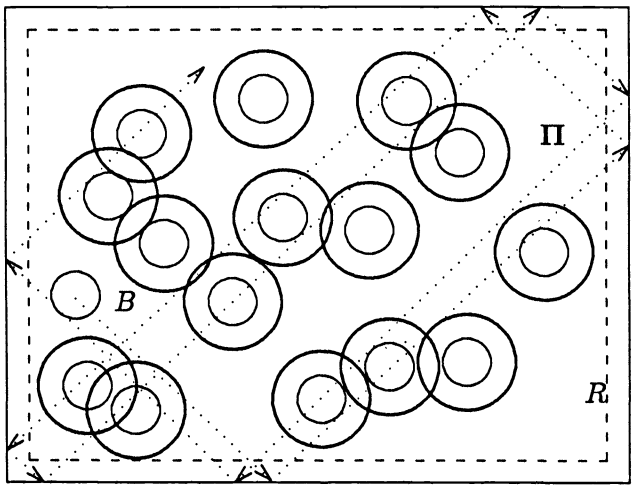


FIG. 8. *Expanding the circles and shrinking the table.*

Hence, we can solve the above problem by applying our general algorithm for red-blue intersection detection. Since the arcs in Γ are closed curves such that no two of them intersect in more than two points, and the arcs in Γ' intersect in at most one point, it follows from Theorem 4.5 that we can detect an intersection between Γ and Γ' in time

$$O((n\sqrt{m\alpha(m)}+m\sqrt{n})\log^{3/2}(m+n)).$$

However in this problem the arcs in Γ and Γ' have some additional properties that improve the running time of the algorithm. For example the segments in Γ' form a connected path, so Γ' can be partitioned into subsets of appropriate size with the property that the boundary of every face in the arrangement of each subset is connected. Hence by the remark following Theorem 4.5, the running time of the above algorithm is $O(m\sqrt{n}\log^{3/2} n)$.

On the other hand, observe that B collides with any other obstacle if and only if the unbounded face $\mathcal{F}$ of $\mathcal{A}(\Gamma)$ and Π intersect. Kedem et al. [KLPS86] have proved that the complexity of $\mathcal{F}$ is $O(n)$, and that it can be computed in time $O(n\log^2 n)$; as a matter of fact, this can be improved to $O(n\log n)$, using the Voronoi diagram of the centers of the circles (see [OY85], [LS87]). Since the edges in $\mathcal{F}$ are nonintersecting, by the remark following Theorem 4.5 we can detect an intersection between $\mathcal{F}$ and Γ' in time $O(m\sqrt{n}\log n + n\log n)$.

Finally, the circles in Γ have the same radius, Corollary 4.11 implies that we can detect an intersection between Γ and Γ' in randomized expected time $O((m+n)^{4/3+\varepsilon})$, for any $\varepsilon > 0$. Therefore, we can conclude Theorem 5.1.

THEOREM 5.1. *The "billiard ball" problem defined above can be solved*:

(i) *Deterministically in time* $O(m\sqrt{n}\log n + n\log n)$;

(ii) *In randomized expected time* $O((m+n)^{4/3+\varepsilon})$, *for any* $\varepsilon > 0$.

Remark 5.2. We can extend our deterministic algorithm in several ways. For example, we can generalize our algorithm for balls with different radii. In fact our algorithm works for balls of other shapes as well.

Another application, more related to computer graphics, is the following multiple ray-shooting, or the "death squad" problem:

> Given m rays in the plane and a collection of objects bounded by n algebraic arcs (of small degree), does any ray hit any object?

Again this problem can be reduced to an instance of the red-blue intersection detection problem by considering the boundaries of the objects as a collection of red arcs and the rays as a collection of blue arcs. Hence, assuming the objects are nonintersecting, we can solve this problem in (deterministic) time $O(m\sqrt{n}\log n)$ by Remark 4.6(ii) following Theorem 4.5, or in randomized expected time

$$O(n^{2/3-\varepsilon}m^{2/3+2\varepsilon} + m\alpha(m)\log^2 m\log n),$$

for any $\varepsilon > 0$, using Corollary 4.12.

If instead of rays we shoot identical bullets of some convex shape having small and fixed complexity, we expand each object by the bullet shape to reduce the problem to the original ray shooting problem. In this case we can detect an intersection, using our general algorithm, in time $O((m\sqrt{\lambda_{s+2}(n)} + n\sqrt{m\alpha(m)})\log^{1.5}(m+n))$, where s is the maximum number of intersections between any two expanded objects. Again, the running time can be improved by exploiting several special properties of the problem structure. Note that the arcs in Γ' are rays rather than segments. It has been proved in [ABP88] that a single face in an arrangement of m rays has only $O(m)$ complexity and can be calculated in $O(m\log m)$ time, therefore the running time becomes $O((m\sqrt{\lambda_{s+2}(n)}\log^{1.5}(m+n) + n\sqrt{m}\log(m+n))$. Finally, if all the objects being shot at are convex, and all bullets are identical and convex, any pair of arcs in Γ intersect in at most two points (cf. [KLPS86]), and the union of all expanded objects can be computed in time $O(n\log n)$. As in the billiard-ball problem it suffices to detect an intersection between the unbounded face of $\mathcal{A}(\Gamma)$ and Γ', therefore we can detect an intersection in additional $O(m\sqrt{n}\log n)$ time. Hence, we can conclude Theorem 5.3.

THEOREM 5.3. (i) *The multiple ray shooting can be solved deterministically in*

$$O(m \cdot \sqrt{n} \cdot \log n + n \log (m+n))$$

time, or in randomized expected time

$$O(n^{2/3-\varepsilon} m^{2/3+2\varepsilon} m\alpha(m) \log^2 m \log n).$$

(ii) *The multiple bullet shooting problem can be solved deterministically in time*

$$O(m\sqrt{\lambda_{s+2}(n)} \log^{1.5} (m+n) + n\sqrt{m} \log (m+n)),$$

which reduces to

$$O(m\sqrt{n} \log n + n \log n)$$

when all objects are convex.

Remark 5.4. Our techniques also allow us to solve efficiently various other extensions of the above problem. For example, if the bullets being shot have a finite range, then we need to detect an intersection between a collection of segments and the given objects. Similarly, in the case of a "stone-throwing squad," where all the stones move along parabolic trajectories in, say, the $x-z$ plane, or along other (algebraic) trajectories, we need to detect an intersection between those trajectories and the objects.

6. Conclusions. In this paper we have obtained several efficient algorithms for the red-blue intersection detection problem. Our algorithm for the case when each of $\mathscr{A}(\Gamma)$ and $\mathscr{A}(\Gamma')$ is a connected planar graph is close to optimal, and our general (deterministic and randomized) algorithms are faster than previously known algorithms. We have applied these algorithms to obtain fast algorithms for several problems in motion planning and collision detection. However, there are still several open questions, as listed below:

(i) The most important question is whether the running time of our deterministic algorithm for the general case can be improved. Theorem 4.10 shows that an efficient algorithm for computing many faces in the arrangement of a given set of Jordan arcs also yields a fast algorithm for the red-blue intersection detection problem, therefore one way of improving the running time is to find an efficient deterministic algorithm to compute many faces in an arrangement of Jordan arcs.

(ii) We have also obtained randomized algorithms for several special cases that are faster than our deterministic algorithm, but at present we do not have any such algorithm for the general case.

(iii) Another open question is whether a red-blue intersection can be detected in randomized expected time $O((n+m) \log^{O(1)} (m+n))$ if all the arcs are line segments. This appears to be a very difficult problem because it is closely related to Hopcroft's problem that calls for detecting an incidence between a set of points and a set of lines, for which no solution better than roughly $(m+n)^{4/3}$ (see [Ag89], [EGS88], [GOS88]) is known.

(iv) Finally, can we report all red-blue intersections in an output-sensitive fashion for general arcs as well?

Since the submission of this paper, there have been further developments that imply certain improvements on some of the results obtained in this paper. Recently, Agarwal [Ag89] has given a deterministic algorithm that counts red-blue intersections for line segments in time $O((m+n)^{4/3} \log^4 (m+n) \log \log (m+n))$ or, more generally, reports all red-blue intersections in the same time with $O(1)$ overhead per intersection. In the further special case where there is no "red-red" or "blue-blue" intersection

between the segments, Chazelle et al. [CEGS88] have given an $O((m+n)\log(m+n))$ algorithm to count all red-blue intersections.

Acknowledgments. We thank the referees for their helpful comments that improved the presentation of this paper.

REFERENCES

[Ag89] P. AGARWAL, *An efficient deterministic algorithm for partitioning arrangements of lines and its applications*, in Proc. 5th ACM Symposium on Computational Geometry, 1989, pp. 11–22.

[AShSh87] P. AGARWAL, M. SHARIR, AND P. SHOR, *Sharp upper and lower bounds on the length of Davenport–Schinzel sequences*, Tech. Report 332, Department of Computer Science, New York University, 1987; J. Combin. Theory Ser A, to appear.

[ABP88] P. ALEVIZOS, J. BOISSONNAT, AND F. PREPARATA, *On the boundary of a union of rays*, manuscript, 1988.

[At85] M. ATALLAH, *Some dynamic computational geometry problems*, Comput. Math. Appl., 11 (1985), pp. 1171–1181.

[BO79] J. BENTLEY AND T. OTTMANN, *Algorithms for reporting and counting intersections*, IEEE Trans. Comput., 28 (1979), pp. 643–647.

[Ch85] B. CHAZELLE, *Fast searching in a real algebraic manifold with applications to geometric complexity*, in Proc. CAAP'85, Lecture Notes in Computer Science, Springer-Verlag, Berlin 1985, pp. 145–156.

[CE88] B. CHAZELLE AND H. EDELSBRUNNER, *An optimal algorithm for intersecting line segments in the plane*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 590–600.

[CEGS88] B. CHAZELLE, H. EDELSBRUNNER, L. GUIBAS, AND M. SHARIR, *Algorithms for bichromatic line segment problems and polyhedral terrains*, manuscript, 1988.

[CG89] B. CHAZELLE AND L. GUIBAS, *Visibility and intersection problems in plane geometry*, Discrete Comput. Geometry, 4 (1989).

[CS88] B. CHAZELLE AND M. SHARIR, *An algorithm for generalized point location and its application*, Tech. Report 369, Department of Computer Science, New York University, May 1988; J. Symb. comput., to appear.

[Cl87] K. CLARKSON, *New applications of random sampling in computational geometry*, Discrete Comput. Geometry, 2 (1987), pp. 195–222.

[Cl89] ———, *New applications of random sampling in computational geometry* II, Discrete Comp. Geometry, 4 (1989), pp. 387–422.

[CEGSW88] K. CLARKSON, H. EDELSBRUNNER, L. GUIBAS, M. SHARIR, AND E. WELZL, *Combinatorial complexity bounds for arrangements of curves and surfaces*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 568–579.

[Co36] R. COURANT, *Differential and integral calculus*, Vol. II, Interscience Publishers, New York, 1936.

[EGH*89] H. EDELSBRUNNER, L. GUIBAS, J. HERSHBERGER, R. SEIDEL, M. SHARIR, J. SNOEYINK, AND E. WELZL, *Implicitly representing arrangements of lines or segments*, Discrete Comp. Geometry, 4 (1989), pp. 433–466.

[EGS88] H. EDELSBRUNNER, L. GUIBAS, AND M. SHARIR, *The complexity of many faces in arrangement of lines and of segments*, in Proc. 4th ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1988, pp. 44–55.

[GHLST87] 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.

[GOS88] L. GUIBAS, M. OVERMARS, AND M. SHARIR, *Intersecting line segments, ray shooting and other applications of geometric partitioning techniques*, Tech. Report RUU-CS-88-26, Department of Computer Science, University of Utrecht, Utrecht, the Netherlands, August 1988.

[GSS89] L. GUIBAS, M. SHARIR, AND S. SIFRONY, *On the general motion planning problem with two degrees of freedom*, Discrete Comp. Geometry, 4 (1989), pp. 491–522.

[HS86] S. HART AND M. SHARIR, *Nonlinearity of Davenport–Schinzel sequences and of generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151–177.

[HW87] D. HAUSSLER AND E. WELZL, *ε-nets and simplex range queries*, Discrete Comput. Geometry, 2 (1987), pp. 127–151.

[KLPS86] K. KEDEM, R. LIVNE, J. PACH, AND M. SHARIR, *On the union of Jordan regions and collision-free translational motion admidst polygonal obstacles*, Discrete Comput. Geometry, 1 (1986), pp. 59–71.

[LS87] D. LEVEN AND M. SHARIR, *Planning a purely translational motion for a convex object in two-dimensional space using generalized Voronoi diagrams*, Discrete Comput. Geometry, 2 (1987), pp. 9–31.

[Lo61] E. LOCKWOOD, *A Book of Curves*, Cambridge University Press, Cambridge, 1961.

[MY86] S. MADDILA AND C. YAP, *Moving a polygon around the corner in a corridor*, in Proc. 2nd ACM Symposium on Computational Geometry, Association for Computing Machinery, New York, 1988, pp. 187–192.

[MS88] H. MAIRSON AND J. STOLFI, *Reporting and counting intersections between two sets of line segments*, NATO ASI Series, F40, Theoretical Foundations of Computer Graphics and CAD, R. Earnshaw, ed., Springer-Verlag, Berlin, 1988, pp. 307–325.

[Mu88] K. MULMULEY, *A fast planar partition algorithm*, I, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 580–589.

[OY85] C. Ó'DÚNLAING AND C. YAP, *A "retraction" method for planning the motion of a disk*, J. Algorithms, 6 (1985), pp. 104–111.

[PSS88] R. POLLACK, M. SHARIR, AND S. SIFRONY, *Separating two simple polygons by a sequence of translations*, Discrete Comput. Geometry, 3 (1988), pp. 123–135.

[Pr79] F. PREPARATA, *A note on locating a set of points in a planar subdivision*, SIAM J. Comput., 8 (1979), pp. 542–545.

[SS87] J. T. SCHWARTZ AND M. SHARIR, *On the two-dimensional Davenport–Schinzel problem*, Tech. Report 193 (revised), Department of Computer Science, New York University, New York, July, 1987; J. Symb. Comput., to appear.

[SH76] M. SHAMOS AND D. HOEY, *Geometric intersection problems*, in Proc. 17th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1976, pp. 208–215.

[Ta83] R. E. TARJAN, *Data Structure and Network Algorithms*, CBMS–NSF Regional Conference Series in Applied Mathematics 44, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1983.

[VdW70] B. VAN DER WAERDEN, *Algebra*, Vol. 2, Frederick Unger, New York, 1970.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 322-328, April 1990

008

TOWARDS AN ARCHITECTURE-INDEPENDENT ANALYSIS OF PARALLEL ALGORITHMS*

CHRISTOS H. PAPADIMITRIOU† AND MIHALIS YANNAKAKIS‡

Abstract. A simple and efficient method for evaluating the performance of an algorithm, rendered as a directed acyclic graph, on any parallel computer is presented. The crucial ingredient is an efficient approximation algorithm for a particular scheduling problem. The only parameter of the parallel computer needed by our method is the message-to-instruction ratio τ. Although the method used in this paper does not take into account the number of processors available, its application to several common algorithms shows that it is surprisingly accurate.

Key words. multiprocessing, parallel computation, communication delay, approximation, scheduling, dag

AMS(MOS) subject classification. 68Q25

1. Introduction. Harnessing the massively parallel architectures, soon to become available, into efficient algorithmic cooperation is one of the most important intellectual challenges facing computer science today. To the theoretician, the task seems similar to that of understanding the issues involved in the performance of sequential algorithms (which motivated Knuth's books, among other important works), only infinitely more complex. In sequential computation the design process involves (a) choosing an algorithm and (b) analyzing it (mostly, counting its steps). In the parallel context, however, we have at least four stages: (1) Choose the algorithm (say, a directed acyclic graph (dag) indicating the elementary computations and their interdependence, a model in which evaluation of sequential performance is trivial). (2) Choose a particular multiprocessor architecture. (3) Find a *schedule* whereby the algorithm is executed on the computer. (4) Only now can we talk about the performance of the algorithm, measured in elapsed time for computing the last result. In our opinion, it is this multilayered nature of the problem that lies at the heart of the difficulties encountered in the development of the necessary ideas, principles, and tools for the design of parallel algorithms.

Are there ways to shortcut the process, thus improving our chances of finally understanding parallel algorithms? The challenge here is to combine stages (2), (3), and (4) into a single step whereby the performance of the algorithm chosen in (1) can be evaluated in a simple, direct manner (at least in principle), pretty much as it is done in sequential computation (stage (b)). At first, the task seems impossible, since it is well known that the performance of a parallel algorithm depends critically on the architecture adopted in stage (2), and the space of architectures is too rich to use as a parameter. A first attempt at this problem was made in [PU]. In that paper three parameters of an algorithm-dag that are relevant in any architecture were isolated, and thus can be used as first measures of the performance of the algorithm (these parameters were as follows: elapsed time ignoring communication, total communication traffic, and total communication delay). In [PU] nontrivial trade-offs between these

* Received by the editors January 10, 1989; accepted for publication July 25, 1989.

† Department of Computer Science and Engineering, University of California, San Diego, California 92093. The research of this author was supported by the National Science Foundation under grant CCR8704170.

‡ AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, New Jersey 07974-2070.

parameters were shown for the pyramid (called "diamond dag" in [PU]). Those results were not completely satisfactory for at least two reasons: First, [PU] bypassed the problem of parametrizing architectures by considering many performance measures. Second, no general principle or technique, applicable to all dags, was in sight (this was mentioned as a challenging open problem in [PU]; the results presented here apply to all dags, and, when restricted to the pyramid, imply Theorem 2 in [PU], thus resolving that question).

In this paper we present a technique that, we feel, may lead to an important new understanding of parallel computation. We are concerned with stages (2), (3), and (4) alone. In other words, we start with the algorithm-dag given and fixed; usually, the dag will in fact be a family thereof, indexed by its number of inputs, such as the pyramid, the full binary tree, the fast Fourier transform (FFT), etc. We assume that we are interested in the case in which the available number of processors is adequate for dealing with the whole width of the dag (thus the number of processors involved is no longer a parameter). It would seem then that the crucial measure of the complexity of the dag would be its *depth*, since it is the depth that dominates its parallel complexity in idealized models of computation, such as a parallel random access machine (PRAM). The point is that in real parallel architectures there is a significant *communication delay* between the time some information is produced at a processor and the time it can be used by another, and depth fails to capture this delay. Such communication delay, denoted τ, is measured in units of elementary processor-steps (equivalently, nodes of the dag).

The status of τ as the single most important parameter of parallel computers was pointed out by Bell [Be]. As a function of n, the number of processors, τ can range, depending on the architecture, from n (on a ring) to $\sqrt{n}$ (a grid) to $\log n$ (a hypercube). For actual computers, one hears of values of τ ranging from small constants (smaller than one in the case of GF11 [Be]) to several millions for slow clusters. We propose that τ is a parameter of the architecture that is important enough to be the basis of our scheme.

Once we have parametrized away the choice of the architecture, we must concentrate on stage (3), scheduling the dag on the processors. But given τ, and under our assumption of enough processors, this is a concrete scheduling problem: Schedule a dag of unit time tasks on an unbounded number of processors so that, if task i is executed at time t on processor p and j is a parent of i, then either (a) j is executed on p by time $t-1$, or (b) j is executed on some other processor p' by time $t-1-\tau$. The optimum makespan of this scheduling problem, always a function of τ, is therefore a fair measure of the parallel complexity of the dag. Unfortunately, this is a nontrivial function; the scheduling problem is NP-complete.

Fortunately, in this paper we propose a simple way to calculate the optimum makespan *within a factor of two.* We compute a simple function e on the nodes of the dag, visiting them depth-first, and show that this quantity is *between half the optimum makespan and the optimum makespan*, and thus can serve as an adequate estimate.

There may be a healthy suspicion that perhaps our method, by disregarding the number of processors, will yield unrealistic, processor-wasteful algorithms. We present strong evidence that this is not so: Using our method, we derive asymptotic upper and lower bounds for the parallel complexity of common algorithms (the FFT, the pyramid, the full binary tree), as a function of n, the number of inputs, and τ. Interestingly, these bounds are achieved by time-optimal algorithms that are also *processor optimal*

(for fixed time), and, furthermore, they are *the most obvious and intuitive parallel algorithms* for the corresponding problem on a parallel machine with ratio τ. They are all simple variants of the algorithm directly suggested by our method. In fact, starting from our lower bound for the pyramid, we can derive the execution time—communication delay trade-off in [PU].

2. The approximation algorithm. We are given a directed acyclic graph $D=(V, A)$, whose nodes are computational tasks of equal (unit) execution time requirements, and whose arcs, as usual in parallel algorithms, represent both time precedence and functional dependence. We are also given a positive integer τ. A *schedule* S of D is a finite set of triples $S \subset V \times \omega \times \omega$ (the second component is the processor on which the node is scheduled, and the third is the time), such that the following conditions hold: (1) For each $v \in V$ there is at least one triple $(v, p, t) \in S$. (2) There are no two triples (v, p, t), $(v', p, t) \in S$, $v \neq v'$. (3) If $(u, v) \in A$ and $(v, p, t) \in S$, then either there is another triple $(u, p, t') \in S$ with $t' \leqq t-1$, or there is another triple $(u, p', t') \in S$ with $t' \leqq t-1-\tau$.

In other words, we schedule the dag with possible repetitions of the nodes on an unlimited number of processors, so that there is a delay τ for communication between the processors. Our goal is to minimize $T_{\max}$, the largest time appearing in S.

NP-completeness.

THEOREM 1. *It is an* NP*-complete problem to decide, given a directed acyclic graph* (V, A), *integer* τ, *and deadline* $T_{\max}$, *whether there exists a schedule* S *such that no time greater than* $T_{\max}$ *is used.*

Proof. The problem is obviously in NP. NP-completeness follows by a reduction from CLIQUE. We are given a graph $G=(V, E)$ and an integer k. We may assume that G has at least $\binom{k}{2}$ edges, because otherwise G does not have a clique of size k. We shall construct a dag $D=(U, A)$, τ and $T_{\max}$, such that D can be scheduled within time $T_{\max}$ if and only if G has a clique of size k.

D is constructed as follows: For each $v \in V$, we have in D a path $(d_{v1}, \cdots, d_{v,|V|^2})$, with all arcs $(d_{vi}, d_{v,i+1})$, $i=1, \cdots, |V|^2-1$. Also, for each edge $e=[u, v] \in E$, there is in D a node c_e. Finally, there is also a node t in D. As for the arcs of D, there is an arc $(d_{v,|V|^2}, c_e)$ from the last node of the path corresponding to v to any node corresponding to an edge e, where $v \in e$. Finally, for all $e \in E$, there is an arc (c_e, t). We define $\tau=|V|^2(k-1)+\binom{k}{2}$, and $T_{\max}=|V|^2k+|E|$. We claim that there is a clique of size k in G if and only if there is a feasible schedule.

Suppose that there is a feasible schedule, and suppose that node $c_{[u,v]}$ executes at processor p before time $d=|V|^2k+\binom{k}{2}$. Since there is not enough time for any of the chains corresponding to u and v to have executed in another processor with the results sent to p, it follows that p has executed both of these chains. On the other hand, t must be executed at the same processor as all edges computed after or at time d. This implies that there must be at most $|E|-\binom{k}{2}$ such edges. Each of the $\binom{k}{2}$ remaining edges must be executed on the same processor that executed the paths of their nodes. Since there is already not enough time to send the results, all these edges, and also the remaining ones, and then t, must be executed on the same processor, which means that this processor must execute at most $T_{\max}-|E|/|V|^2=k$ paths. However, this in turn implies that there are k nodes in V that are adjacent to $\binom{k}{2}$ edges, and thus a clique of size k exists.

Conversely, if a clique of size k exists, a feasible schedule is the following: Execute each path corresponding to a node not in the clique in a different processor, and execute on the same processor the paths corresponding to nodes in the clique, followed

by the edges between these nodes, then (exactly as all other messages arrive from the other processors) the remaining edges, and finally t. □

Note that this proof establishes that the same scheduling problem with recomputation not allowed is also NP-complete. This is because in the *if* direction we computed each task once.

The Approximation Algorithm. Given a dag (V, A) and an integer τ, we can compute the following function e: $V \to \omega$, inductively on the depth of the nodes of the dag:

(1) If v is a source, then $e(v) = 0$.

(2) Otherwise, consider the set of all ancestors of v, ordered in decreasing $e(u)$ (recall that e has already been evaluated on the ancestors) $e(u_1) \geqq e(u_2) \geqq \cdots \geqq e(u_p)$. Let k be the smaller of $\tau + 1$ and p. Then we define $e(v) = e(u_k) + k$.

LEMMA 1. *There is no schedule in which node v is scheduled before time $e(v)$.*

Proof. By induction on the depth of v. The result certainly holds if v is a source. For the induction step, suppose that v can be scheduled at time $t < e(v)$, and consider the k ancestors $u_1, \cdots, u_k$ of v that have the highest e value (recall the definition of k in this context). By the induction hypothesis, each of these nodes u_i was scheduled on or after time $e(u_i) \geqq e(u_k)$. Since $t < e(v) = e(u_k) + k \leqq e(u_k) + \tau + 1$, each of these nodes u_i is scheduled less than $\tau + 1$ before v, and thus must be scheduled on the same processor as v. Since there are k such nodes scheduled between times $e(u_k)$ and t, we must have $t \geqq e(u_k) + k$, or $t \geqq e(v)$, contradicting our assumption. □

LEMMA 2. *For each node v there is a schedule in which node v is scheduled at time $2e(v)$.*

Proof. Again by induction on the depth of v. Clearly, any source can be scheduled at $t = 0$. For the induction step, suppose first that v has $p \leqq \tau + 1$ ancestors. Then, we can schedule all of them, including v, by time $e(v) + 1 \leqq 2e(v)$ on the same processor.

Thus, assume that v has $p > \tau + 1$ ancestors. Order them in decreasing e value: $e(u_1) \geqq e(u_2) \geqq \cdots \geqq e(u_p)$. Since $e(v) = e(u_{\tau+1}) + \tau + 1$, we have that $e(v) - \tau - 1 \geqq e(u_j)$ for $j \geqq \tau + 1$. Thus, by induction, all ancestors of v except for the τ first ones can be scheduled (by separate schedules and in different processors) by time $2e(v) - 2\tau - 2$. This means that starting at time $2e(v) - \tau - 1$ the τ first nodes can also be scheduled (in reverse order one after the other), and hence v can be finally scheduled by time $2e(v)$. □

The proof of Lemma 2 suggests the following simple algorithm for scheduling the nodes of the dag, so that every node v is executed by time $2e(v)$: the processor that computes node v computes the τ highest (in e value) ancestors of v and receives the rest from communication from other processors. Combining the lemmata, we have the following theorem.

THEOREM 2. *The above algorithm is an approximation algorithm for $T_{\max}$ with a worst-case ratio of two.*

A generalization. By a rather complicated generalization of the algorithm, we can obtain schedules with the same worst-case approximation ratio even when each node $v \in V$ has an execution time $x(v)$ and a delay $\tau(v)$; that is, if $(v, u) \in A$, u cannot start until $x(v) + \tau(v)$ time units have passed after v started at another processor, or until $x(v)$ time units after v started at the same processor. This models a situation in which tasks have unequal processing times, and their results differ in size, and therefore in transmission delay. In our original problem, all $x(v)$'s are one, and all $\tau(v)$ are equal (and equal to τ).

The following generalization of our algorithm computes the estimate $e(v)$ for this case: For any source node v, $e(v)$ is zero, as before. For any node v other than a

source consider the set of its ancestors, and for each such ancestor u compute the function $f(u) = e(u) + x(u) + \tau(u)$. Now sort the ancestors in decreasing f: $f(u_1) \geqq f(u_2) \geqq \cdots$ (notice that f may not be monotonic with respect to the dag).

Consider an integer j (intuitively, this is a candidate value for $e(v)$), and suppose that $f(u_k) > j \geqq f(u_{k+1})$. Let $N_j(v)$ be the subdag of D consisting of all nodes u_i, $i \leqq k$ for which there is a path to v using only nodes u_i, $i \leqq k$. Consider then the following scheduling problem S_j for the nodes of $N_j(v)$ (excluding v): The release time for job v_i is $e(v_i)$, where we have assumed that $e(v_1) \geqq e(v_2) \geqq \cdots \geqq e(v_l)$, and $N_j(v)$ contains l nodes (plus v). Obviously, the optimum schedule has length $L_j = \max_{i=1}^{k} [e(v_i) + \sum_{q=1}^{i} x(q)]$. *Now choose the least j such that $j \geqq L_j$. This is the value of $e(v)$.*

As in the previous case, we can use here also the e values to schedule the dag. The algorithm is as follows. For every node v, the processor that computes v, computes itself the nodes in $N_{e(v)}(v)$, and receives the other ancestors of v from communication from other processors. The nodes of $N_{e(v)}(v)$ are executed by the processor of v as soon as they become available, where a node u becomes available when the processor knows every parent of u either by computing it itself or from communication from another processor. If several nodes of $N_{e(v)}(v)$ are available at the same time, then the processor may choose an arbitrary one for execution.

THEOREM 3. *The above algorithm is a polynomial-time approximation algorithm for the generalized problem with a worst-case ratio of two.*

Proof. We shall show first that no node can be scheduled before its e value. We use induction on the depth of a node. Suppose that node v is scheduled at time $j < e(v)$, but all of its ancestors are scheduled on or after their e value. By the definition of $e(v)$, there is no way that all the nodes in $N_j(v)$ can be scheduled at the same processor as v (since $L_j > j$). On the other hand, if (u, w) is an arc of $N_j(v)$, and w is scheduled at the same processor as v (before or at time j), then the same must be true of u because $f(u) > j$. Since every node $u \in N_j(v)$ has a path to v through $N_j(v)$, all nodes of $N_j(v)$ must be scheduled at the same processor as v. It follows that no node can be scheduled before its e value.

We next show that our algorithm executes every node v at or before time $2e(v)$. The proof is again by induction on the depth of v. Any node $w \notin N_{e(v)}(v)$ that is a parent of a node u in $N_{e(v)}(v)$ has $f(w) \leqq e(v)$, and thus can be scheduled so that its message reaches the processor of v at time $2e(w) + x(w) + \tau(w) \leqq e(v) + e(w) \leqq e(v) + e(u)$. Thus, the assumption in the scheduling problem $S_{e(v)}$ that a task $u \in N_{e(v)}(v)$ is available for execution at time $e(u)$ is valid, except for an additional delay of $e(v)$ time units. Thus, by shifting an optimum schedule for $S_{e(v)}$, with makespan $e(v)$, $e(v)$ time units to the right, we obtain a valid schedule in which v is executed at time $2e(v)$. Obviously, the one-machine scheduling problem with release times is solved optimally by any algorithm that is greedy, i.e., executes some node as long as there is one available. It follows that our algorithm schedules node v by time $2e(v)$.

To show that the algorithm is polynomial, note that we repeat it for all $v \in V$, and for each v and value of j we compute $N_j(v)$ and L_j; these computations take time $|A| + |V| \log |V|$. There are only $|V|$ values of $f(v)$, and thus an equal number of interesting values of j, for which we must repeat the computation of $N_j(v)$ and L_j. The result is an $O(|V|^2(|A| + |V| \log |V|))$-time algorithm. With a more careful implementation, we can save a factor of $|V|$. □

If there are communication delays on the *arcs* (not nodes) of the dag, the same technique yields a ratio of two. The only difference is that, instead of the criterion based on the function f, $N_j(v)$ is now constructed arc by arc, where an arc (u, w) is

included in the subgraph if $e(u)+x(u)+\tau(u, w)>j$, and we keep only those nodes that can reach v in this subgraph. The rest remains the same.

3. Applications to concrete problems. In this section we apply our technique to three well-known families of dags: the full binary tree, the fast Fourier transform, and the pyramid.

Full binary tree. Suppose that the dag is a full binary tree with n nodes. Recall that our algorithm entails scheduling on the same processor with v its τ highest (in e value) ancestors. In the binary tree, these ancestors are at the $\log \tau$ next levels of the tree. Therefore, in this case our algorithm degenerates to the following: Divide the $\log n$ levels of the tree into $\log n/\log \tau$ layers of height $\log \tau$. Compute each of the n/τ resulting subtrees, with all subtrees at the same level computed in parallel. The time is $O(\tau \log n/\log \tau)$; since it is the result of our method, it is asymptotically optimal. $O(n/\tau)$ processors are used. The number of processors can be made optimal (number of nodes divided by time) by simply letting the last layer of the tree have height $\log (\tau \log n/\log \tau)$, instead of $\log \tau$. The time at most doubles.

The fast Fourier transform. From the point of view of each output, the n-input FFT is a full binary tree with n leaves, and thus the optimal time is, by the previous subsection, $O(\tau \log n/\log \tau)$. The proof of Theorem 2 suggests an algorithm that computes each output separately.

However, the following obvious algorithm achieves the same bound: For $k \approx \tau/\log \tau$, partition the FFT dag into stripes of height $\log k$. Each stripe contains n/k FFTs on k points. Each of these FFTs is executed in sequential time $O(k \log k) \approx \tau$ by a processor, with all FFTs in a stripe executed in parallel. Once a stripe is completed, the results are exchanged and the next stripe starts. The total time is $O(\tau \log n/\log \tau)$ (optimal), and there are $O(n \log \tau/\tau)$ processors (also optimal!).

The pyramid. Arguing in the pyramid with n nodes along the same lines as for the full binary tree, the optimal time bound is obtained by computing, in the same processor as node v, the subpyramid with τ nodes with v as apex. The time required is $2\sqrt{n\tau}$. This method uses $\sqrt{n}-\sqrt{\tau}$ processors. However, the *stripes* method [PU], in which each processor computes a diagonal stripe of width $\sqrt{\tau}$ and processors are synchronized in a pipelined fashion, gives the same asymptotic time and optimum number of processors: $\sqrt{n/\tau}$.

We can now reconstruct the idealized time-communication delay trade-off, shown in [PU], as follows: Recall that the idealized time T of a schedule is the time required if all delays are zero, and the communication delay d is the time required if all execution times are zero and $\tau=1$. It was shown in [PU] that (in our notation), for the n-node pyramid, $Td=\Omega(n)$. We need the following lemma, relating d and T with our $T_{\max}$ for any dag D.

LEMMA 3. *If there is a schedule S with idealized time T and communication delay d, then there is a schedule S' with $T_{\max}=T+d\tau$.*

Proof. We transform a legal schedule S under [PU] to a legal schedule S' in our framework. As in [PU], assign a delay to each slot of each processor. If a slot of processor P executes a node u, and u is a source, then the delay is 0; if u is not a source, the delay is the smallest integer i such that for every parent w of u, either there is an earlier slot of P that executes w with delay at most i, or there is an earlier slot of another processor that executes w with delay at most $i-1$. The idealized delay d of S is the maximum delay of any slot. For every processor P of S and every delay i, we have a processor P_i in S' that executes the slots of P that have delay i, shifted $i\tau$

time units later. It is easy to see that S' is a legal schedule in our framework, and its makespan is $T+d\tau$. Thus, $T_{\max} \leqq T+d\tau$. □

Continuing our argument for the pyramid with n nodes, since our 2-approximate algorithm schedules the pyramid within time $2\sqrt{n\tau}$, we know that for any τ we must have $T_{\max} \geqq \sqrt{n\tau}$. Combining this inequality with Lemma 3 we conclude that $T+d\tau \geqq \sqrt{n\tau}$. Picking a τ such that $T=\frac{1}{2}\sqrt{n\tau}$, we have that $d\tau \geqq \frac{1}{2}\sqrt{n\tau}$, from which $Td \geqq \frac{1}{4}n$ follows.

4. Open problems. Is there a polynomial-time approximation algorithm for the scheduling problem with worst-case ratio better than two? We mention that, in the case with no recomputation, another NP-completeness proof establishes that no approximation ratio better than two is possible (unless P = NP). But, then again, we know of no approximation algorithm at all for the problem without recomputation. If τ is a fixed integer (or even a fraction with a fixed numerator), Jungen, Kirousis, and Spirakis (1988) showed that a dynamic programming approach yields a polynomial-time algorithm (with the numerator of τ in the exponent of the polynomial) for solving the scheduling problem *exactly*.

Although our approximation algorithm is bound to produce schedules with many processors and a lot of recomputation, we observed that, in all examples considered, the same bounds are achieved without recomputation. Is this a general pattern, namely, that any parallel algorithm using recomputation can be simulated by an algorithm that does not use recomputation and has the same asymptotic time requirements? This is an interesting question, akin to open problems proposed in [PU]. One can easily produce dags with large out-degrees, for which recomputation is indeed necessary, so the interesting question is the one in which the degrees of the dags are bounded, say by two. Jungen, Kirousis, and Spirakis (1988) observed that the *inverse full binary tree* (directed from the root to the leaves) requires logarithmic recomputation, on the average.

Finally, since e, the central part of our method, is a rather simple parameter of the dag, there is hope that we can develop the necessary algebraic tools to show tight bounds for the parallel complexity of *problems* (*not* algorithms), e.g., for the discrete Fourier transform, as opposed to the FFT, and matrix multiplication, as opposed to the full binary tree, etc. Liu (1988) has recently shown that the $\Omega(\tau \log n/\log \tau)$ lower bound for the full binary tree holds for any tree with n nodes, and thus for any algorithm for adding, multiplying, etc., n indeterminants, $n \times n$ matrix multiplication, etc.

Acknowledgment. Many thanks to Gene Lawler for pointing out to us another approximation algorithm (with a larger worst-case ratio) for the problem of the subsection entitled "A Generalization," thus motivating that subsection.

REFERENCES

[Be] C. G. BELL, *Gordon Bell on the future of computers*, SIAM News, 20(2), March 1987, pp. 1, 21.

[JKS] H. JURGEN, L. KIROUSIS, AND P. SPIRAKIS, Lower bounds and efficient algorithms for multiprocessor scheduling of dags with communication delays, Proc. 1989 ACM Symposium on Parallel Algorithms and Architectures, Santa Fe, NM, 1989, pp. 254–264.

[Liu] D. LIU, manuscript, University of California, San Diego, CA, 1988.

[PU] C. H. PAPADIMITRIOU AND J. D. ULLMAN, A Communication-Time tradeoff", Proc. 1984 STOC.; SIAM J. Comput., 16 (1987), pp. 639–646.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 329-340, April 1990

009

ADDITION MACHINES*

ROBERT W. FLOYD† AND DONALD E. KNUTH†

Abstract. It is possible to compute gcd (x, y) efficiently with only $O(\log xy)$ additions and subtractions, when three arithmetic registers are available but not when there are only two. Several other functions, such as x^y mod z, are also efficiently computable in a small number of registers, using only addition, subtraction, and comparison.

Key words. addition, subtraction, greatest common divisor, exponentiation, Fibonacci numbers, RSA encryption, lower bounds

AMS(MOS) subject classification. 68Q05

An addition machine is a computing device with a finite number of registers, limited to the following six types of operations:

read x	{input to register x}
$x \leftarrow y$	{copy register y to register x}
$x \leftarrow x+y$	{add register y to register x}
$x \leftarrow x-y$	{subtract register y from register x}
if $x \geqq y$	{compare register x to register y}
write x	{output from register x}.

The register contents are assumed to belong to a given set A, which is an additive subgroup of the real numbers. If A is the set of all integers, we say the device is an *integer addition machine*; if A is the set of all real numbers, we say the device is a *real addition machine.*

We will consider how efficiently an integer addition machine can do operations such as multiplication, division, greatest common divisor, exponentiation, and sorting. We will also show that any addition machine with at least six registers can compute the ternary operation $x\lfloor y/z \rfloor$ with reasonable efficiency, given $x, y, z \in A$ with $z \neq 0$.

Remainders. As a first example, consider the calculation of

$$x \bmod y = \begin{cases} x - y\lfloor x/y \rfloor, & \text{if } y \neq 0; \\ x, & \text{if } y = 0. \end{cases}$$

This binary operation is well defined on any additive subgroup A of the reals, and we can easily compute it on an addition machine as follows:

```
P1:   read x; read y; z ← z − z;
      if y ≧ z then
          if z ≧ y then {y = 0, do nothing}
          else if x ≧ z then while x ≧ y do x ← x − y
                  else repeat x ← x + y until x ≧ z
      else if z ≧ x then while y ≧ x do x ← x − y
                  else repeat x ← x + y until z ≧ x;
      write x.
```

* Received by the editors April 10, 1989; accepted for publication (in revised form) August 2, 1989. This research was supported in part by the National Science Foundation under grant CCR-86-10181, and by Office of Naval Research contract N00014-87-K-0502.

† Computer Science Department, Stanford University, Stanford, California 94305.

(There is implicitly a finite-state control. A pidgin Pascal program such as this one is easily converted to other formalisms; cf. [1].)

Program P_1 handles all sign combinations of x and y; therefore it is rather messy. In the special case where $x \geqq 0$ and $y > 0$, a much simpler program applies:

```
P2:  read x; read y;  {assume that x ≧ 0 and y > 0}
     while x ≧ y do x ← x − y;
     write x.
```

Any program for this special case can be converted to a program of comparable efficiency for the general case by using the identities

$$-x = (x - x) - x;$$

$$(-x) \bmod (-y) = -(x \bmod y);$$

$$(-x) \bmod y = \begin{cases} y - (x \bmod y), & \text{if } x \bmod y \neq 0; \\ 0, & \text{if } x \bmod y = 0. \end{cases}$$

General programs for multiplication, division, and the greatest common divisor (gcd) can be constructed similarly from algorithms that assume positive operands. We shall therefore restrict consideration to positive cases in the algorithms below.

Program P_2 performs $\lfloor y/x \rfloor$ subtractions. Can we do better? Yes; here, for example, is a program that uses a doubling procedure to subtract larger multiples of y:

```
P3:  read x; read y;      {assume that x ≧ 0 and y > 0}
     while x ≧ y do
          begin z ← y;
          repeat w ← z; z ← z + z until not x ≧ z;
          x ← x − w;
          end;
     write x.
```

This program repeatedly subtracts $w = 2^k y$ from x, where $k = \lfloor \log_2 (x/y) \rfloor$; thus, it implicitly computes the binary representation of $\lfloor x/y \rfloor$, from left to right. The total running time is bounded by $O(\log (x/y))^2$, which is considerably smaller than $\lfloor x/y \rfloor$ when $\lfloor x/y \rfloor$ is large.

Further improvement, to a running time that is $O(\log (x/y))$ instead of $O(\log (x/y))^2$, appears at first sight to be impossible, because an addition machine has only finitely many registers and it cannot divide by 2. Therefore the numbers y, $2y$, $4y$, $8y$, ... must all apparently be computed again and again if we want to use a trick based on doubling.

A Fibonacci method. Remainders can, however, be computed with the desired efficiency $O(\log (x/y))$ if we implicitly use the Fibonacci representation of $\lfloor x/y \rfloor$ instead of the binary representation. Define Fibonacci numbers as usual by

$$F_0 = 0; \qquad F_1 = 1; \qquad F_n = F_{n-1} + F_{n-2}, \quad \text{for } n \geqq 2.$$

Every nonnegative integer n can be uniquely represented [9] in the form

$$n = F_{l_1} + F_{l_2} + \cdots + F_{l_t}, \qquad l_1 \gg l_2 \gg \cdots \gg l_t \gg 0,$$

where $t \geqq 0$ and $l \gg l'$ means that $l - l' \geqq 2$. If $n > 0$, this representation can be found by first choosing l_1 such that

$$F_{l_1} \leqq n < F_{l_1+1},$$

so that $n - F_{l_1} < F_{l_1+1} - F_{l_1} = F_{l_1-1}$, and then by writing

$$n = F_{l_1} + (\text{Fibonacci representation of } n - F_{l_1}).$$

We shall let

$$\lambda n = l_1 \quad \text{and} \quad \nu n = t$$

denote respectively the index of the leading term and the number of terms in the Fibonacci representation of n. By convention, $\lambda 0 = 1$.

Fibonacci numbers are well suited to addition machines because we can go from the pair $\langle F_l, F_{l+1}\rangle$ up to the next pair $\langle F_{l+1}, F_{l+2}\rangle$ with a single addition, or down to the previous pair $\langle F_{l-1}, F_l\rangle$ with a single subtraction. Furthermore, Fibonacci numbers grow exponentially, about 69% as fast as powers of 2. They have been used as power-of-2 analogues in a variety of algorithms (see, for instance, "Fibonacci numbers" in the index to [3]) and Matijasevich's solution to Hilbert's tenth problem in [6]).

If we let two registers of an addition machine contain the pair of numbers $\langle yF_l, yF_{l+1}\rangle$, where l is an implicit parameter, it is easy to implement the operations

$$l \leftarrow 1, \qquad l \leftarrow l+1, \qquad l \leftarrow l-1$$

and to test the conditions

$$x \geqq yF_l, \qquad x < yF_{l+1}, \qquad l = 1.$$

Therefore we can compute $x \bmod y$ efficiently by implementing the following procedure:

```
read x; read y;      {assume that x ≧ 0 and y > 0}
if x ≧ y then
        begin l ← 1;
        repeat l ← l + 1 until x < yF_{l+1};
        repeat if x ≧ yF_l then x ← x − yF_l;
            l ← l − 1;
        until l = 1;
        end;
write x.
```

The first **repeat** loop increases l until we have

$$yF_l \leqq x < yF_{l+1},$$

i.e., until $l = \lambda n$, where $n = \lfloor x/y \rfloor$. The second loop decreases l while subtracting

$$yF_{l_1} + yF_{l_2} + \cdots + yF_{l_t} = yn$$

from x according to the Fibonacci representation of n. The result, $x - ny = x \bmod y$, has been computed with

$$2\lambda n - 2 + \nu n = O(\log (x/y))$$

additions and subtractions altogether.

Here is the same program expressed directly in terms of additions and subtractions, using only three registers:

```
P4:   read x; read y;        {assume that x ≧ 0 and y > 0}
      if x ≧ y then
          begin z ← y;
          repeat ⟨y, z⟩ ← ⟨z, y + z⟩ until not x ≧ z;        {x ≧ y still holds}
          repeat if x ≧ y then x ← x − y;
                 ⟨y, z⟩ ← ⟨z − y, y⟩;
          until y ≧ z;
          end;
      write x.
```

The multiple assignment "$\langle y, z\rangle \leftarrow \langle z, y+x\rangle$" is an abbreviation for the operation "set $y \leftarrow y+z$ and interchange the roles of registers y and x in the subsequent program"; the assignment "$\langle y, z\rangle \leftarrow \langle z-y, y\rangle$" is similar. By making two copies of this program code, in one of which the variables y and z are interchanged, we can jump from one copy to the other and obtain a legitimate addition-machine program; cf. [4, Ex. 7].

A formal proof of correctness for program P_4 would establish the invariant relation

$$\exists l \geqq 1 \quad (y = y_0 F_l \quad \text{and} \quad z = y_0 F_{l+1})$$

in the case $x_0 \geqq y_0$, where x_0 and y_0 are the initial values of x and y.

Multiplication and division. We can use essentially the same idea to compute the ternary operation $x\lfloor y/z\rfloor$ efficiently on any addition machine. This time we accumulate multiples of x as we discover the Fibonacci representation of $\lfloor y/z\rfloor$:

```
read x; read y; read z;      {assume that y ≧ 0 and z > 0}
w ← 0;
if y ≧ z then
        begin l ← 1;
        repeat l ← l + 1 until not y ≧ z F(l+1);
        repeat if y ≧ z F(l) then ⟨w, y⟩ ← ⟨w + x F(l), y − z F(l)⟩;
             l ← l − 1;
        until l = 1;
        end;
write w.
```

The actual addition-machine code requires six registers, because we need Fibonacci multiples of x as well as z:

```
P5:   read x; read y; read z;   {assume that y ≧ 0 and z > 0}
      w ← w − w;
      if y ≧ z then
          begin u ← x; v ← z;
          repeat ⟨u, x⟩ ← ⟨x, u + x⟩; ⟨v, z⟩ ← ⟨z, v + z⟩;
          until not y ≧ z;      {y ≧ v still holds}
          repeat if y ≧ v then ⟨w, y⟩ ← ⟨w + u, y − v⟩;
               ⟨u, x⟩ ← ⟨x − u, u⟩; ⟨v, z⟩ ← ⟨z − v, v⟩;
          until v ≧ z;
          end;
      write w.
```

The key invariant relations, in the case $y_0 \geqq z_0$, are now

$$\exists l \geqq 1 \quad (u = x_0 F_l, \quad x = x_0 F_{l+1}, \quad v = z_0 F_l, \quad z = z_0 F_{l+1});$$
$$\exists n \geqq 0 \quad (w = x_0 n, \quad y = y_0 - z_0 n).$$

If we suppress x, u, and w from this program, the **repeat** statements act on $\langle y, v, z\rangle$ exactly as the **repeat** statements in our previous program act on $\langle x, y, z\rangle$. Therefore, if $y_0 \geqq z_0$, we have $y = y_0 \bmod z_0 = y_0 - z_0\lfloor y_0/z_0\rfloor$ after the **repeat** statements in the new program. Hence $w = x_0\lfloor y_0/z_0\rfloor$ as desired. The total number of additions and subtractions is

$$4\lambda n - 3 + 2\nu n = O(\log(y_0/z_0)),$$

where $n = \lfloor y_0/z_0\rfloor$.

An integer addition machine can make use of the constant 1 by reading that constant into a separate, dedicated register. Then we can specialize the ternary algorithm by setting $z \leftarrow 1$ (for multiplication) or $x \leftarrow 1$ (for division). Thus we can compute the product xy in $O(\log \min(|x|, |y|))$ operations and the quotient $\lfloor y/z\rfloor$ in $O(\log|y/z|)$ operations, using only addition, subtraction, and comparison of integers. (Multiplication and division clearly cannot be done unless such constants are used, since any function $f(x, y, \ldots)$ computed by an addition machine that inputs the sequence of values $\langle x, y, \ldots\rangle$ must satisfy $f(\alpha x, \alpha y, \ldots) = \alpha f(x, y, \ldots)$ for all $\alpha > 0$.)

Greatest common divisors. Euclid's algorithm for the greatest common divisor of two positive integers x and y can be formulated as follows:

read x; **read** y; {assume that $x > 0$ and $y \geqq 0$}
while $y > 0$ **do** $\langle x, y\rangle \leftarrow \langle y, x \bmod y\rangle$;
write x.

The **while** loop preserves the invariant relation $\gcd(x, y) = \gcd(x_0, y_0)$. After the first iteration, we have $x > y \geqq 0$; the successive values of x are strictly decreasing and positive, so the algorithm must terminate.

We can therefore use our method for computing $x \bmod y$ to calculate $\gcd(x, y)$ on an integer addition machine:

```
P6:  read x; read y;        {assume that x > 0 and y ≧ 0}
     z ← y; z ← z + z;
     while not y ≧ z do {equivalently, while y > 0, since z = 2y}
          begin while x ≧ z do ⟨y, z⟩ ← ⟨z, y + z⟩;
          repeat if x ≧ y then x ← x − y;
               ⟨y, z⟩ ← ⟨z − y, y⟩;
          until y ≧ z;
          ⟨x, y⟩ ← ⟨y, x⟩; z ← y; z ← z + z;
          end;
     write x.
```

(Here the operation $\langle x, y\rangle \leftarrow \langle y, x\rangle$ should not really be performed; it means that the roles of registers x and y should be interchanged. The implementation jumps between six copies of this program, one for each permutation of the register names x, y, z.)

This algorithm will compute $\gcd(x, y)$ correctly on a general addition machine, whenever the ratio y/x is rational. Otherwise, it will loop forever.

The total number of operations performed by program P_6 is

$$T(x, y) = f(q_1) + f(q_2) + \cdots + f(q_m) + 6,$$

where $q_1, q_2, \ldots, q_m$ is the sequence of quotients $\lfloor x/y \rfloor$ in the respective iterations of Euclid's algorithm, and where $f(q)$ counts the number of operations in one iteration of the outermost **while** loop. If $q = 0$ (this case can occur only on the first iteration), we have one assignment, one addition, one subtraction, and four comparisons; so $f(0) = 7$. If $q > 0$ we have one assignment, $\lambda q - 1$ additions, $\lambda q + \nu q - 1$ subtractions, and $3\lambda q - 2$ comparisons; so

$$f(q) = 5\lambda q + \nu q - 3.$$

We have $f(1) = 8$, $f(2) = 13$, and, in general, $f(F_l) = 5l - 2$ for all $l \geqq 2$.

This three-register algorithm for greatest common divisor turns out to be quite efficient, even though it uses only addition, subtraction, and comparison. Indeed, the numbers in the registers never exceed 2 max (x, y), where x and y are the given inputs, and we can obtain rather precise bounds on the running time.

THEOREM 1. *Let* $N = \max(x, y)/\gcd(x, y)$. *The number of operations* $T(x, y)$ *performed by program* P_6 *satisfies*

$$3 \log_\phi N + \alpha \leqq T(x, y) \leqq 13.5 \log_\phi N + \beta,$$

for some constants α *and* β, *where* $\phi = (1 + \sqrt{5})/2$.

Proof. We can assume that $x > y$; then all the q's are positive. If $F_l \leqq q < F_{l+1}$, we have $\lambda q = l$ and $1 \leqq \nu q \leqq l/2$, hence

$$5l - 2 \leqq f(q) \leqq 5.5l - 3.$$

Furthermore, we have $\phi^{l-2} \leqq F_l \leqq \phi^{l-1}$; hence

$$5 \log_\phi (q + 1) - 2 \leqq f(q) \leqq 5.5 \log_\phi q + 8.$$

Summing over all values $q_1, \ldots, q_m$ gives

$$5 \log_\phi ((q_1 + 1) \cdots (q_m + 1)) - 2m \leqq T(x, y) - 6 \leqq 5.5 \log_\phi (q_1 \cdots q_m) + 8m.$$

Now let the values occurring in Euclid's algorithm be $x_0, x_1, \ldots, x_{m+1}$, where $x_0 = x$, $x_1 = y$, $x_{j+1} = x_{j-1} \bmod x_j$, $x_m = \gcd(x, y)$, and $x_{m+1} = 0$. Then $q_j = \lfloor x_{j-1}/x_j \rfloor$ for $1 \leqq j \leqq m$, and we have

$$q_1 q_2 \cdots q_m \leqq \frac{x_0}{x_1} \frac{x_1}{x_2} \cdots \frac{x_{m-1}}{x_m} < (q_1 + 1)(q_2 + 1) \cdots (q_m + 1).$$

The product $(x_0/x_1)(x_1/x_2) \cdots (x_{m-1}/x_m) = x_0/x_m$ is just what we have called N. Furthermore, we have $m \leqq \log_\phi N$ by a well-known theorem of Lamé [2, Thm. 4.5.3F]. This suffices to complete the proof. □

When the inputs are consecutive Fibonacci numbers $\langle x, y \rangle = \langle F_m, F_{m+1} \rangle$ with $m \geqq 2$, we have $q_1 = 0$, $q_2 = \cdots = q_{m-1} = 1$, $q_m = 2$, and the total running time is

$$T(F_m, F_{m+1}) = 7 + 8(m - 2) + 13 + 6 = 8m + 10.$$

This appears to be the worst case, in the sense that it seems to maximize $T(x, y)$ over all pairs $\langle x, y \rangle$ with $\max(x, y) \leqq F_{m+1}$. Computations for small n support this conjecture, which (if true) would imply that the upper bound in Theorem 1 could be improved to $8 \log_\phi N + \beta$.

Stacks. Euclid's algorithm defines a one-to-one correspondence between pairs of relatively prime positive integers $\langle x, y \rangle$ with $x > y$ and sequences of positive integers $\langle q_1, \ldots, q_m \rangle$, where each $q_j \geqq 1$ and $q_m \geqq 2$. We can push a new integer q onto the front of such a sequence by setting $\langle x, y \rangle \leftarrow \langle qx + y, x \rangle$; we can pop $q_1 = \lfloor x/y \rfloor$ from the front by setting $\langle x, y \rangle \leftarrow \langle y, x \bmod y \rangle$.

Therefore an integer addition machine can represent a stack of arbitrary depth in two of its registers. The operation of pushing or popping a positive integer q can be done with $O(\log q)$ operations, using a few auxiliary registers.

Here, for example, is the outline of an integer addition program that reads a sequence of positive integers followed by zero and writes out those positive integers in reverse order:

$\langle x, y\rangle \leftarrow \langle 2, 1\rangle$; {the empty stack}
repeat read q;
  if $q \geqq 1$ **then** $\langle x, y\rangle \leftarrow \langle qx + y, x\rangle$;
until not $q \geqq 1$;
repeat $\langle q, x, y\rangle \leftarrow \langle \lfloor x/y \rfloor, y, x \bmod y\rangle$;
  if $y \geqq 1$ **then write** q;
until not $y \geqq 1$.

This program uses the algorithms for multiplication and division shown earlier. The running time to reverse the input $\langle q_1, q_2, \ldots, q_m, 0\rangle$ is $O(m + \log q_1 q_2 \ldots q_m)$.

We can sort a given list of positive integers $\langle q_1, q_2, \ldots, q_m\rangle$ in a similar way, using the classical algorithms for merge sorting with three or more magnetic tapes that can be "read backwards" [3, § 5.4.4]. The basic operations required are essentially those of a stack, so we can sort in $O((m + \log q_1 q_2 \cdots q_m) \log m)$ steps if there are at least 12 registers.

Exponentiation. We can now show that an integer addition machine is able to compute

$$x^y \bmod z$$

in $O((\log y)(\log z) + \log (x/z))$ operations. The basic idea is simple: We first form the numbers

$$x_l = x^{F_l} \bmod z$$

for $2 \leqq l \leqq \lambda y$; this requires one multiplication mod z for each new value of l, once $x_2 = x \bmod z$ has been found in $O(\log (x/z))$ operations. Then we use the Fibonacci representation of y to compute $x^y \bmod z$ with $\nu y - 1$ further multiplications mod z. For example, $x^{11} \bmod z$ is computed by successively forming the powers

$$x^1, \quad x^2, \quad x^3, \quad x^5, \quad x^8, \quad x^{8+3}$$

modulo z.

There is, however, a difficulty in carrying out this plan with only finitely many registers, since the method we have used to discover the Fibonacci representation of y determines the relevant terms F_l in reverse order from the way we need to calculate the relevant factors x_l.

One solution is to push the numbers $x_2, x_3, \ldots, x_{\lambda y}$ onto a simulated stack as they are being computed. Then we can pop them off in the desired order as we discover the Fibonacci representation of y. Each stack operation takes $O(\log z)$ time, since each x_l is less than z; hence the stacking and unstacking requires only $O((\log y)(\log z))$ operations, and the overall running time changes by at most a constant factor.

But the stacking operation forms extremely large integers, having $\Theta((\log y)(\log z))$ bits, so it is not a practical solution if we are concerned with the size of the numbers being added and subtracted as well as the number of additions and subtractions. An algorithm that needs only $O((\log y)(\log z))$ additions and subtractions of integers that never get much larger than z would be far more useful in practice.

We can obtain such an algorithm if we first compute the "Fibonacci reflection" of y, namely, the number

$$y^R = F_{2+\lambda y-l_1} + F_{2+\lambda y-l_2} + \cdots + F_{2+\lambda y-l_t}$$

when y has the Fibonacci representation

$$y = F_{l_1} + F_{l_2} + \cdots + F_{l_t}.$$

Then we can use the Fibonacci representation of y^R to determine the relevant factors x_l as we compute them; no stack is needed.

Here is a program that computes y^R, assuming that $y>0$ and that both y and the constant 1 have already been read into registers named y and 1.

$u \leftarrow 1;\ v \leftarrow 1;\ w \leftarrow y;$ $\quad \{u = F_l, v = F_{l+1}, l = 1\}$
repeat $\langle u, v\rangle \leftarrow \langle v, u+v\rangle$ **until not** $w \geqq v;$ $\quad \{u = F_l, v = F_{l+1}, y \geqq u\}$
$\quad \{u = F_l, v = F_{l+1}, l = \lambda y\}$
$r \leftarrow 1;\ s \leftarrow 1;\ t \leftarrow t - t;$
repeat if $w \geqq u$ **then**
$\quad$ **begin** $w \leftarrow w - u;\ t \leftarrow t + s;$
$\quad$ **end;**
$\quad \langle u, v\rangle \leftarrow \langle v - u, u\rangle;\ \langle r, s\rangle \leftarrow \langle s, r+s\rangle;$ $\quad \{l \leftarrow l - 1\}$
until $u \geqq v$.

Throughout this program we have $u = F_l$ and $v = F_{l+1}$, where l begins at 1, rises to λy, and returns to 1. During the second **repeat** statement we have also

$$r = F_{1+\lambda y-l}, \qquad s = F_{2+\lambda y-l}, \qquad t = (y-w)^R.$$

The program terminates with $l = 1$ and $w = 0$; hence we have

$$r = F_{\lambda y}, \qquad s = F_{\lambda y+1}, \qquad t = y^R.$$

The full program for $x^y \bmod z$ can now be written as follows, using routines described earlier:

read x; **read** y; **read** z;
$\langle r, s, t\rangle \leftarrow \langle F_{\lambda y}, F_{\lambda y+1}, y^R\rangle;$
$x \leftarrow x \bmod z;\ w \leftarrow x;\ u \leftarrow 1;$ $\quad \{x = x_l, w = x_{l+1}, l = 1\}$
repeat if $t \geqq r$ **then**
$\quad$ **begin** $t \leftarrow t - r;\ u \leftarrow (uw) \bmod z;$
$\quad$ **end;**
$\quad \langle r, s\rangle \leftarrow \langle s - r, r\rangle;\ \langle x, w\rangle \leftarrow \langle w, (xw) \bmod z\rangle;$ $\quad \{l \leftarrow l + 1\}$
until $r \geqq s$;
write u.

The invariant relations

$$x = x_l, \qquad w = x_{l+1}, \qquad r = F_{1+\lambda y-l}, \qquad s = F_{2+\lambda y-l}$$

are maintained in the final **repeat** loop as l increases from 1 to λy.

For example, if $y = 11 = 8+3 = F_6 + F_4$, we have $\lambda y = 6$ and $y^R = F_2 + F_4 = 1+3 = 4$. Hence $r = 8$, $s = 13$, $t = 4$, $u = 1$, and $x = w = x_0 \bmod z_0$ at the beginning of the final **repeat**. The registers will then contain the following respective values at the moments when the final **until** statement is encountered:

r	s	t	u	x	w
5	8	4	1	$x_0 \bmod z_0$	$x_0^2 \bmod z_0$
3	5	4	1	$x_0^2 \bmod z_0$	$x_0^3 \bmod z_0$
2	3	1	$x_0^3 \bmod z_0$	$x_0^3 \bmod z_0$	$x_0^5 \bmod z_0$
1	2	1	$x_0^3 \bmod z_0$	$x_0^5 \bmod z_0$	$x_0^8 \bmod z_0$
1	1	0	$x_0^{11} \bmod z_0$	$x_0^8 \bmod z_0$	$x_0^{13} \bmod z_0$

The statement "$u \leftarrow (uw) \bmod z$" can be implemented by first forming uw and then taking the remainder mod z, using the multiplication and division algorithms

presented earlier. But we can do better by changing the multiplication algorithm so that the quantities being added together for the final product are maintained modulo z: We simply change appropriate operations of the form $\alpha \leftarrow \alpha + \beta$ to the sequence

$\alpha \leftarrow \alpha + \beta$;
if $\alpha \geqq z$ **then** $\alpha \leftarrow \alpha - z$.

Then the register contents never get large. In fact, if x_0 and y_0 are initially nonnegative and less than z_0, all numbers in the algorithm will be nonnegative and less than $2z_0$. We have proved the following result:

THEOREM 2. *If* $0 \leqq x, y < z \leqq 2^{n-1}$, *the quantity* $x^y \bmod z$ *can be computed from* x, y, *and* z *with* $O((\log y)(\log z))$ *additions and subtractions of integers in the interval* $[0 \,..\, 2^n)$, *on a machine with finitely many registers.*

Indeed, the constant implied by this O is reasonably small. The algorithm just sketched may therefore find practical application in the design of special-purpose hardware for $x^y \bmod z$, which is the fundamental operation required by the RSA scheme of encoding and decoding messages [7].

Lower bounds. Some of the algorithms presented above can be shown to be optimal, up to a constant factor. For example, we obviously need $\Omega(\log \min(x, y))$ additions to compute the product xy; we cannot compute any number larger than $2^k \max(x, y)$ with k additions, and if $2^k < \min(x, y)$ this is less than $\min(x, y) \max(x, y) = xy$.

Logarithmic time is also necessary for division and gcd, even if we extend addition machines to *addition-multiplication machines* (which can perform multiplication as well as addition in one step). An elegant proof of this lower bound was given by Stockmeyer in an unpublished report [8]. We reproduce his proof here for completeness.

THEOREM 3 (Stockmeyer). *An integer addition-multiplication machine requires* $\Omega(\log x)$ *arithmetic operations to compute* $\lfloor x/2 \rfloor$, $x \bmod 2$, *or* $\gcd(x, 2)$, *for infinitely many* x.

Proof. If we can compute $\lfloor x/2 \rfloor$ or $\gcd(x, 2)$ in t steps, we can compute $x \bmod 2 = x - 2\lfloor x/2 \rfloor = 2 - \gcd(x, 2)$ in at most $t+2$ steps. So it suffices to prove that $x \bmod 2$ requires $\Omega(\log x)$ steps.

Any computation of an integer addition-multiplication machine on a given input x forms polynomials in x and compares polynomial values. A t-step computation defines at most 2^t different *computation paths*, depending on the results of **if** tests. For convenience we assume that each statement of the form "**write** w" is changed to

if $0 \geqq w$ **then write** w **else write** w.

Then a program that computes $x \bmod 2$ must take a different path when x is changed to $x+1$.

Each computation path is defined by a sequence of polynomial tests

$$q_1(x) : 0, \qquad q_2(x) : 0, \qquad \ldots, \qquad q_s(x) : 0$$

made at times $t_1 < t_2 < \cdots < t_s \leqq t$. (Different paths have different polynomials in general, although $q_1(x)$ will be the same on each path.) If $q_j(x)$ corresponds to a test at time t_j, the degree of $q_j(x)$ is at most $2^{t_j - 1}$. Therefore the sum of the degrees of the $q_j(x)$ is less than 2^t. Therefore the total number of roots of all the polynomials $q_j(x)$, taken over all computation paths of length t, is less than 2^{2t}.

Let m be the least integer $\geqq 2^{2t}$ such that none of the polynomials described in the previous paragraph has a root in the closed interval $[m, m+1]$. Each root can exclude at most two values of m; therefore $m \leqq 2^{2t} + 2^{2t+1}$. By definition, the addition-multiplication program takes the same computation path when it is applied to $x = m$

and to $x = m+1$; therefore it does not compute $x \bmod 2$ on both of these values. Therefore there is an integer x_t in the interval $[2^{2^t}, 2^{2^{t+2}})$ such that the value $x_t \bmod 2$ has not been computed at time t on any of the computation paths. Therefore there are infinitely many x for which the time to compute $x \bmod 2$ is $\Omega(\log x)$.

So far we have counted both arithmetic operations and conditional tests as steps of the computation. This also gives a lower bound on the number of arithmetic operations, since we can assume without loss of generality that no computation path makes more than $\binom{k}{2}$ consecutive conditional tests when there are k registers. This completes the proof. □

Notice that Stockmeyer's argument establishes the lower bound $\Omega(\log x)$ on the total computation time, even if the number of registers is unbounded, and even if the programs are allowed to introduce arbitrary constants. A straightforward generalization of the proof shows that an integer addition-multiplication machine needs $\Omega(\log(x/y))$ steps to compute $x \bmod y$, uniformly for all $y > 0$ and for infinitely many x when y is given. However, the argument does not apply to machines with unbounded registers and indirect addressing; for this case Stockmeyer [8] used a more complex argument to obtain the lower bound $\Omega(\log x/\log\log x)$. It is still unknown whether indirect addressing can be exploited to do better than $O(\log x)$. When integer division is allowed, as well as addition and multiplication, the bound $\Omega(\log\log\log \min(x, y))$ on arithmetic operations needed to compute $\gcd(x, y)$ has been proved by Mansour, Schieber, and Tiwari [5].

Our efficient constructions have all been for addition machines that contain at least three registers. The following theorem shows that 2-register addition machines cannot do much.

THEOREM 4. *Any algorithm that computes* $\gcd(x, y)$ *on an integer addition machine with only two registers needs* $\Omega(n-1)$ *operations to compute* $\gcd(n, 1)$.

LEMMA. *Consider a graph on unordered pairs* $\{x, y\}$ *of nonnegative integers, where* $\{x, y\}$ *is adjacent to* $\{x, x+y\}$, $\{x+y, y\}$, $\{|x-y|, y\}$, *and* $\{x, |x-y|\}$. *The shortest path from* $\{n, 1\}$ *to* $\{1, 1\}$ *in this graph has length* $n-1$, *for all* $n \geqq 1$.

Proof of the Lemma (by Tomás Feder). Consider the following four operations on unordered pairs $\{x, y\}$:

$\underline{A}$. Replace $\min(x, y)$ by $x+y$.
$\bar{A}$. Replace $\max(x, y)$ by $x+y$.
$\underline{S}$. Replace $\min(x, y)$ by $\max(x, y) - \min(x, y)$.
$\bar{S}$. Replace $\max(x, y)$ by $\max(x, y) - \min(x, y)$.

Then $\underline{A}\bar{S} = \bar{A}\bar{S} = \underline{S}\underline{S} = \text{identity}$ and $\underline{S}\bar{S} = \bar{S}$. Furthermore, $\underline{S}$ is either $\bar{S}\underline{A}$ or $\bar{S}\bar{A}$, hence $\underline{A}\underline{S}$ and $\bar{A}\underline{S}$ are either $\underline{A}$ or $\bar{A}$. Any minimal sequence of operations must therefore begin with $\bar{S}$'s and end with A's. But $\bar{S}^k$ applied to $\{n, 1\}$ yields $\{n-k, 1\}$, for $k < n$; and A's do not decrease anything. Therefore the shortest path is $\bar{S}^{n-1}$. □

Proof of Theorem 4. As in the proof of Theorem 3, the sequence of **if** tests made by an addition machine defines a computation path, dependent on the inputs. We say that the test "**if** $x \geqq y$" is *critical* if it is performed at a moment when the contents of registers x and y happen to be identical.

Let M be a 2-register addition machine that produces the output $M(a, b)$ when applied to inputs $\langle a, b\rangle$. We assume that a and b are initially present in the two registers; therefore the computation path corresponding to $\langle a, b\rangle$ will be the computation path corresponding to $\langle ma, mb\rangle$ for all integers $m \geqq 1$.

Every computation path defines constants α and β such that $M(a, b) = \alpha a + \beta b$ for all $\langle a, b\rangle$ leading to this path. If M never encounters a critical test when applied

to $\langle a, b\rangle$, it will follow the same path on inputs $\langle am, bm\rangle$ and $\langle am+1, bm\rangle$ for all sufficiently large values of m. Therefore we will have $M(am+1, bm) = M(am, bm)+\alpha$ for all large m; and M cannot be a valid program for computing the gcd. We have proved that every 2-register gcd program must make a critical test before it produces an output.

Next we show that every 2-register gcd machine must make a critical test before it uses any instruction of the form $x \leftarrow x-x$ or $x \leftarrow x+x$. Suppose M performs such an instruction when it is applied to inputs $\langle a, b\rangle$; these inputs determine a computation path defining constants α and β such that the other register, y, contains $\alpha a+\beta b$ when $x \leftarrow x-x$ or $x \leftarrow x+x$ is performed. If no critical tests have occurred, the same computation path will be followed when the inputs are $\langle a^2bm+1, ab^2m\rangle$ and $\langle a^2bm, ab^2m+1\rangle$, for all sufficiently large m. But $\gcd(a^2bm+1, ab^2m) = \gcd(a^2bm, ab^2m+1) = 1$; hence y must contain an odd value when M is applied to $\langle a^2bm+1, ab^2m\rangle$ or $\langle a^2bm, ab^2m+1\rangle$. (If y is even when x is being set to $x-x$ or $x+x$, both registers will contain an even value; hence M cannot subsequently output "1".) Hence $\alpha(a^2bm+1)+\beta(ab^2m)$ and $\alpha(a^2bm)+\beta(ab^2m+1)$ are odd, for all sufficiently large m; hence α and β are both odd. But $\gcd(2a^2bm+1, 2ab^2m+1)$ is odd, and the inputs $\langle 2a^2bm+1, 2ab^2m+1\rangle$ follow the same path as $\langle a, b\rangle$ for all large m; hence $\alpha(2a^2bm+1)+\beta(2ab^2m+1)$ must be odd, a contradiction.

Therefore every 2-register gcd machine must make a critical test, before which it has performed only operations of the forms $x \leftarrow x \pm y$, $y \leftarrow y \pm x$. Such operations correspond to the transformations considered in the lemma.

Suppose M is applied to the inputs $\langle n, 1\rangle$. When the first critical test occurs, we have $x = y$; and $\gcd(x, y) = \gcd(n, 1) = 1$, because $\gcd(x, y)$ is preserved by all of the operations $x \leftarrow x \pm y$ or $y \leftarrow y \pm x$ that have been performed so far. Thus $x = y = \pm 1$; the algorithm must have followed a path from $\{n, 1\}$ to $\{1, 1\}$ in the sense of the lemma. So the algorithm must have performed at least $n-1$ operations before reaching the first critical test. This completes the proof. □

Further restrictions. A "minimalist" definition of addition machines would eliminate the copy operation $x \leftarrow y$, because this operation can be achieved by

$$x \leftarrow x-x; \qquad x \leftarrow x+y.$$

We can also simplify the **if** tests, allowing only the one-register form "**if** $x \geqq 0$", because a general two-register comparison "**if** $x \geqq y$ **then** α **else** β" can be replaced by

$x \leftarrow x-y$;
if $x \geqq 0$ **then begin** $x \leftarrow x+y$; α **end**
else begin $x \leftarrow x+y$; β **end**.

Similarly, we can do away with addition, if we add a new register ξ, because $x \leftarrow x+y$ can be achieved by three subtractions:

$$\xi \leftarrow \xi-\xi; \qquad \xi \leftarrow \xi-y; \qquad x \leftarrow x-\xi.$$

Addition cannot be eliminated without increasing the number of registers, in general. For we can prove that the operation $x_1 \leftarrow x_1+x_2$ cannot be achieved by any sequence of operations of the forms $x_i \leftarrow x_i-x_j$, for $1 \leqq i, j \leqq r$. The proof can be formulated in matrix theory as follows.

Let E_{ij} be the matrix that is all zeros except for a 1 in row i and column j. We want to show that the matrix $I+E_{12}$ cannot be obtained as a product of matrices of the form $I-E_{ij}$. Clearly we cannot use the matrices $I-E_{jj}$, whose determinant is zero;

so we must use $I - E_{ij}$ with $i \neq j$. But the inverse of $I - E_{ij}$ is $I + E_{ij}$, when $i \neq j$. So if

$$I + E_{12} = (I - E_{i_1 j_1}) \cdots (I - E_{i_m j_m})$$

we have, taking inverses,

$$I - E_{12} = (I + E_{i_m j_m}) \cdots (I + E_{i_1 j_1}),$$

which is patently absurd, since the right side contains no negative coefficients.

Open questions.

(1) Can the upper bound in Theorem 1 be replaced by $8 \log_\phi N + \beta$?

(2) Can an integer addition machine with only five registers compute x^2 in $O(\log x)$ operations? Can it compute the quotient $\lfloor y/z \rfloor$ in $O(\log |y/z|)$ operations?

(3) Can an integer addition machine compute $x^y \bmod z$ in $o((\log y)(\log z))$ operations, given $0 \leqq x, y < z$?

(4) Can an integer addition machine sort an arbitrary sequence of positive integers $\langle q_1, q_2, \ldots, q_m \rangle$ in $o((m + \log q_1 q_2 \cdots q_m) \log m)$ steps?

(5) Can the powers of 2 in the binary representation of x be computed and output by an integer addition machine in $o(\log x)^2$ steps? For example, if $x = 13$, the program should output the numbers 8, 4, 1 in some order.

(6) Is there an efficient algorithm to determine whether a given $r \times r$ matrix of integers is representable as a product of matrices of the form $I + E_{ij}$?

Acknowledgment. We wish to thank Baruch Schieber for calling our attention to Stockmeyer's paper [8].

REFERENCES

[1] D. E. KNUTH AND R. H. BIGELOW, *Programming languages for automata*, J. Assoc. Comput. Mach., 14 (1967), pp. 615-635.

[2] D. E. KNUTH, *The Art of Computer Programming, Vol. 2: Seminumerical Algorithms*, Addison-Wesley, Reading, MA, 1969.

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

[4] ———, *Structured programming with* **goto** *statements*, Comput. Surveys, 6 (1974), pp. 261-301.

[5] Y. MANSOUR, B. SCHIEBER, AND P. TIWARI, *Lower bounds for integer greatest common divisor computations*, Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988, pp. 54-63.

[6] Y. V. MATIJASEVICH, *Enumerable sets are diophantine*, Dokl. Akad. Nauk SSSR, 191 (1970), pp. 279-282; Soviet Math. Dokl., 11 (1970), pp. 354-357.

[7] R. L. RIVEST, A. SHAMIR, AND L. ADLEMAN, *A method for obtaining digital signatures and public-key cryptosystems*, Comm. ACM, 21 (1978), pp. 120-126.

[8] L. J. STOCKMEYER, *Arithmetic versus Boolean operations in idealized register machines*, IBM Thomas J. Watson Research Center, Report RC 5954, 21 April 1976.

[9] E. ZECKENDORF, *Représentation des nombres naturels par une somme de nombres de Fibonacci ou de nombres de Lucas*, Bull. Soc. Roy. Sci. Liège, 41 (1972), pp. 179-182.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 341-383, April 1990

010

SEMANTIC CORRECTNESS FOR A PARALLEL OBJECT-ORIENTED LANGUAGE*

J. J. M. M. RUTTEN†

Abstract. Different semantic models are studied for a language called POOL: parallel object-oriented language. It is a simplified version of POOL-T, a language that is actually used to write programs for a parallel machine. The most important aspect of this language is that it describes a system as a collection of communicating objects that all have internal activities which are executed in parallel. For POOL, operational and denotational semantics have been developed previously. The former aims at the intuitive operational meaning of the language, whereas the main characteristic of the latter is compositionality. In this paper, the author relates both models, which are quite different, and proves the semantic correctness of the denotational semantics with respect to the operational semantics. These semantic investigations take place in the mathematical framework of complete metric spaces. For the operational semantics a simple space of functions from states to compact sets of streams (which are sequences of states) is used; for the denotational semantics, a domain of processes is used, which is the solution of a reflexive domain equation over a category of complete metric spaces. The main mathematical tool we use is Banach's theorem, which states that contractions on complete metric spaces have unique fixed points. Both the operational and the denotational semantics are reformulated and are presented, as well as many operators on the semantic domains, as the fixed point of a suitably defined contraction. In this way, a formal equivalence between both models is established. For this purpose, an intermediate domain, which is first compared to the operational model by means of an abstraction operator, is introduced. This function takes processes, which are treelike structures, as arguments and yields sets of streams as results. Next, it is shown that both intermediate and the denotational model are fixed points of the same contraction, from which their equality follows. From both facts, the main result of this study follows: The operational meaning of a POOL program is equal to the denotational meaning to which the abstraction operator is applied. In this manner, the correctness of the denotational semantics with respect to the operational semantics is established.

Key words. operational semantics, denotational semantics, process creation, object-oriented programming, semantic correctness, complete metric spaces, contractions

AMS(MOS) subject classifications. 68B10, 68C01

C.R. classifications. D.3.1, F.3.2, F.3.3

1. Introduction.

We study different semantic models for a language called POOL: parallel object-oriented language. Although the theoretical foundations of object-oriented programming in general, and of *parallel* object-oriented programming in particular, have not been paid much attention to, this language has been extensively studied in a formal semantic context: In [ABKR86(a)] and [ABKR86(b)], an operational and a denotational semantics of POOL have been developed. The main goal of this paper is to compare the two models, which are quite different, by proving some formal relation between them, which at the same time will establish the correctness of the denotational semantics with respect to the operational semantics. Before we explain in some detail the language POOL and the contents of this paper, we first give a short explanation of the notion of semantic correctness and the way it can be proved.

A semantics for a programming language $\mathcal{L}$ is a mapping $\mathcal{M}: \mathcal{L} \to D$, where D is some mathematical domain (a set, a complete partial ordering, a complete metric space), which we call the semantic *universe* of $\mathcal{M}$. Sometimes $\mathcal{M}$ is called a model for $\mathcal{L}$. Traditionally, two main types of semantics are distinguished: *operational* semantics

* Received by the editors October 31, 1988; accepted for publication (in revised form) July 12, 1989. This work was carried out in the context of ESPRIT project 415: Parallel Architectures and Languages for AIP—a VLSI-directed approach.

† Centre for Mathematics and Computer Science, P.O. Box 4079, 1009 AB Amsterdam, the Netherlands.

and *denotational* semantics. Without getting involved in a discussion about the precise definitions, we state that in our view the main characteristic of the former is that its definition is based on a *transition relation* [HP79], [Pl81], [Pl83]; a denotational semantics is characterised by the fact that it is defined in a *compositional* manner: the denotational semantics of a composite statement is given in terms of the denotational semantics of its components. (As a second distinctive property one often considers the way in which recursion is treated: The usual view is that an operational semantics treats recursion by means of so-called *syntactic environments* (or body replacement), whereas a denotational semantics uses *semantic environments*, in combination with some fixed-point argument.)

Now consider an operational semantics $\mathcal{O}: \mathcal{L} \to D$ and a denotational semantics $\mathcal{D}: \mathcal{L} \to D'$. A natural question is whether $\mathcal{D}$ is *correct* with respect to $\mathcal{O}$, that is, whether $\mathcal{D}$ makes *at least* the same distinction as $\mathcal{O}$ does. (Often, $\mathcal{D}$ makes more; see [KR88] for a simple example.) If we define for a semantics $\mathcal{M}: \mathcal{L} \to D''$ an equivalence relation $\equiv_{\mathcal{M}}$ by

$$s \equiv_{\mathcal{M}} t \Leftrightarrow \mathcal{M}[\![s]\!] = \mathcal{M}[\![t]\!],$$

for all $x, t \in \mathcal{L}$, then the correctness of $\mathcal{D}$ with respect to $\mathcal{O}$ can be formally expressed by the condition

$$\equiv_{\mathcal{D}} \subset \equiv_{\mathcal{O}}.$$

One way to prove the correctness of $\mathcal{D}$ is to introduce a so-called *abstraction* operator $\alpha: D' \to D$, which (is, in general, not injective and) relates the denotational semantic universe with the operational one. If one can prove that

$$\mathcal{O} = \alpha \circ \mathcal{D},$$

then a precise relation between $\mathcal{O}$ and $\mathcal{D}$ has been established, which moreover implies the correctness of $\mathcal{D}$ with respect to $\mathcal{O}$.

As a mathematical framework for our semantic descriptions we have chosen *complete metric spaces*. (For the basic definitions of topology see [Du66] or [En77].) In this we follow and generalize [BZ82]. (For other applications of this type of semantic framework see [BKMOZ86].) We follow [KR88] in using contractions on complete metric spaces as our main mathematical tool, exploring the fact that contractions have *unique* fixed points (Banach's theorem). We shall define both operators on our semantic universes and the semantic models themselves as fixed points of suitably defined contractions. In this way, we are able to use a general method for proving semantic correctness: suppose we have defined $\mathcal{O}$ as the fixed point of a contraction

$$\Phi: (\mathcal{L} \to D) \to (\mathcal{L} \to D).$$

If we next show that also $\alpha \circ \mathcal{D}$ is a fixed point of Φ then Banach's theorem implies that $\mathcal{O} = \alpha \circ \mathcal{D}$. Thus complete metric spaces facilitate an equivalence proof that is clearly structured and that, due to the uniqueness of the various fixed points involved, is considerably shorter than it would have been in case, e.g., complete partial orders had been used.

It is the approach sketched above that will be applied to the language POOL. Before doing so, we start in § 2 with a toy language that is extremely simple but has a construct for process creation in common with POOL. This section can be seen as a prolongation of the introduction and tries to give the reader some feeling for the techniques used. Since no definitions or results of this section are used in the other sections it can be skipped without any problem.

The language POOL is described in detail in § 3. It is a simplified version of the language POOL-T, which is defined in [Am85] and for which [Am86] and [Am87] give an account of the design considerations. POOL-T was designed in subproject *A* of ESPRIT project 415 with the purpose of programming a highly parallel machine, which is also being developed in this project (see [Od87] for an overview). The language provides all the facilities needed to program reasonably large parallel systems, and several large applications and many small ones have been written in it.

In POOL, a system is viewed as a collection of *objects*. These are dynamic entities containing *data* (stored in *variables*) and *methods* (a kind of procedures). Objects can be created dynamically during the execution of a program and each of them has an internal activity (its *body*) in which it can execute expressions and statements. While inside an object everything proceeds sequentially, the concurrent execution of the bodies of all the objects can give rise to a large amount of parallelism. Objects can interact by sending *messages* to each other. Acceptance of a message gives rise to a rendez-vous between sender and receiver, during which an appropriate method is executed.

In § 4, we follow [ABKR86(a)] in defining an operational semantics for POOL. It is based on a transition relation and is given, and here we differ from [ABKR86(a)], as the fixed point of a contraction. The semantic domain used is a complete metric space of (functions from states to) compact sets of streams, which are sequences of states.

In § 5, we present a denotational semantics for POOL, very similar to the model given in [ABKR86(b)]. We define a mapping from the set of POOL programs (called *units*) to some reflexive domain of processes $\bar{P}$ (cf. [Pl76]), which is a complete metric space with treelike structures for its elements. It satisfies a reflexive domain equation, which is solved by deriving from it a functor on a category of complete metric spaces and then taking the fixed point of this functor. The mathematical techniques to do so are sketched in § 2 of [ABKR86(b)] and presented in detail in [AR88]. Before we assign a semantic value to the unit as a whole, we first define the semantics of expressions and statements, which will be given by functions of the following type:

$$\mathcal{D}_E : L_E \to AObj \to Cont_E \to \bar{P} \quad \text{and} \quad \mathcal{D}_S : L_S \to AObj \to Cont_S \to \bar{P},$$

where L_E and L_S are the sets of expressions and statements and

$$Cont_E = Obj \to \bar{P}, \qquad Cont_S = \bar{P}.$$

The semantic domain *AObj* stands for the set of (active) object names. Its appearance in the semantics of expressions and statements reflect the fact that in POOL each expression or statement is evaluated *by a certain object*. Further, a *continuation* will be given as an argument to the semantic functions. This describes what will happen *after* the execution of the current expression or statement. As the continuation of an expression generally depends upon the result of this expression (an object name), its type is $Obj \to \bar{P}$, whereas the type of continuations of statements is simply $\bar{P}$. The use of continuations makes it possible to define the semantics, especially of object creation, in a convenient and concise way. (For more examples of the use of continuations in semantics, see [Br86] and [Go79].)

After having defined an operational and a denotational semantics for POOL, we come to the main subject of our paper: The comparison of both models. This constitutes a nontrivial problem, mainly because, first, the respective semantic domains are very different and, second, because the denotational semantics is defined in terms of

continuations, whereas the operational semantics is direct, that is, does not use continuations. Moreover, the communication mechanism of POOL (consisting of message passing with method invocation) is dealt with quite differently by the two models. The solution that we propose consists of the introduction of an intermediate semantic model, in § 6, which has in common with the operational semantics that it is direct (without continuations) and that it is based on the same transition relation, but which has for its range the same reflexive domain of processes as does the denotational model. Then, in § 7, this intermediate model is related to the operational semantics by means of an abstraction operator, which takes processes as arguments and yields sets of streams. Next, it is connected with (an extended version of) the denotational semantics by the observation that both models are fixed point of the same contraction. As a result, it follows that the operational semantics of a unit equals its denotational meaning to which the abstraction operator is applied.

Section 7 is followed by three appendices. Appendix I gives the mathematical definitions we use; in Appendix II, the abstraction operator that is used in the proof of the semantic correctness for POOL is defined in all formal detail. Finally, Appendix III shows how the language POOL can be extended with so-called *standard* objects and how the definitions and proofs can be adapted in order to obtain a similar correctness result for the extended language.

Semantic treatments of parallel object-oriented languages in general are scarce; we only know [Cl81], which gives a detailed mathematical model for an actor language. This is done by defining a set of so-called augmented actor event diagrams, each of which is a fairly complicated structure representing (the beginning of) a single computation. In order to deal with nondeterminism, a novel power domain construction is used. As to the comparison of operational and denotational semantics for languages with process creation, we only know of [AB88], where some simplified versions of POOL are studied. None of these languages, however, contains the original POOL-T constructs for communication (for message passing with method invocation), the treatment of which, in the correctness proof, we consider to be an essential part of this paper.

2. A very simple language with process creation. Before we tackle the main problem of this paper, we would like to start with a much simpler case: We introduce a very small "toy" language L_T and present an operational and a denotational semantics for it. Next, we shall compare these two models. All this can be regarded as a little exercise, a "warming up" so to speak, aiming at a better understanding of what follows in the next section: It turns out that for both the languages L_T and POOL (to be introduced in the next section) the operational and denotational semantics can be compared in very much the same way.

For the definition of L_T we need a set $(a, b \in)A$ of *elementary actions.* (Throughout this paper, we shall use the notation $(x, y \in)X$ for the introduction of a set X with typical elements x and y.) For A we take an arbitrary, possibly infinite, set. It will contain a subset $(c \in)C \subseteq A$ of so-called *communications.* Similarly to CCS [Mil80], we define a bijection $^- : C \to C$ with $^- \circ {}^- = id_C$. It yields for every $c \in C$ a matching communication $\bar{c}$. In $A \setminus C$ we have a special element τ denoting successful communication.

DEFINITION 2.1 (Syntax for L_T). The set of statements $(s, t \in)L_T$ is given by

$$s ::= a | s_1; s_2 | \textbf{new}(s).$$

Note that $a \in A \supseteq C$. To L_T we add a special element E, denoting the *empty* statement. Note that syntactic constructs like $s; E$ and **new** (E) are *not* in L_T.

A statement is of one of the following forms. First, it can be an elementary action a. Here elementary means that it is an uninterpreted action. Examples of possible interpretations are assignments, or read and write actions. Second, a statement s can be the sequential composition $s_1; s_2$ of statements s_1 and s_2. Finally, it may be a new-statement **new**(s), the execution of which amounts to the creation of a new process that executes s. A more detailed explanation will follow below.

The operational semantics will be formulated using the notion of *parallel statements*. A parallel statement is a finite sequence of statements that are to be executed in parallel.

DEFINITION 2.2 (Parallel statements). Let $(\rho, \pi \in) Par$ be given by $Par = (L_T)^*$, the set of finite sequences of statements. Typical elements will also be indicated by $\langle s_1, \cdots, s_n \rangle$, for $n \geqq 1$. For $\rho = \langle s_1, \cdots, s_n \rangle$ and $\pi = \langle t_1, \cdots, t_m \rangle$ we define $\rho \hat{} \pi = \langle s_1, \cdots, s_n, t_1, \cdots, t_m \rangle$.

Next we define the operational semantics of parallel statements. It is based on the well-known notion of a *transition relation* (in the style of Hennessy and Plotkin [HP79], [Pl81], [Pl83].

DEFINITION 2.3 (Transition relation for *Par*). Let $\rightarrow \subseteq Par \times A \times Par$ be the smallest relation (writing $\rho - a \rightarrow \rho'$ for $(\rho, a, \rho') \in \rightarrow$) satisfying:

(1) $\langle a \rangle - a \rightarrow \langle E \rangle$, $\langle a; s \rangle - a \rightarrow \langle s \rangle$,

(2) if $\langle s \rangle - a \rightarrow \rho$, then $\langle \textbf{new}\,(s) \rangle - a \rightarrow \rho$,

(3) if $\langle s, t \rangle - a \rightarrow \rho$, then $\langle \textbf{new}\,(s); t \rangle - a \rightarrow \rho$,

(4) if $\langle s_1; (s_2; s_3) \rangle - a \rightarrow \rho$, then $\langle (s_1; s_2); s_3 \rangle - a \rightarrow \rho$,

(5) if $\rho - a \rightarrow \rho'$, then $\rho \hat{} \pi - a \rightarrow \rho' \hat{} \pi$ and $\pi \hat{} \rho - a \rightarrow \pi \hat{} \rho'$,

(6) if $\rho - c \rightarrow \rho'$ and $\pi - \bar{c} \rightarrow \pi'$, then $\rho \hat{} \pi - \tau \rightarrow \rho' \hat{} \pi'$,

for $a \in A$, $c \in C$, $s, t, s_1, s_2, s_3 \in L_T$, and $\rho, \rho', \pi, \pi' \in Par$.

Intuitively, $\rho - a \rightarrow \rho'$ tells us that starting in the parallel statement ρ the elementary action a can be performed, resulting in the parallel statement ρ'. Interesting in the definition above are (3), (5), and (6). According to (3), the parallel statements $\langle s, t \rangle$ and $\langle \textbf{new}\,(s); t \rangle$ can perform the same elementary actions. In other words, evaluating $\langle \textbf{new}\,(s); t \rangle$ results in a parallel statement $\langle s, t \rangle$. Thus we see that the length of a parallel statement increases when **new**(s) is evaluated. Operationally, this can be viewed as the creation of a process that starts evaluating s, while statement t is being executed in parallel. According to (5), a composite parallel statement $\rho \hat{} \pi$ can perform all the elementary actions that can be performed by either ρ or π. In (6) it is expressed that if ρ can perform a communication action c and π can perform a matching communication action $\bar{c}$, then $\rho \hat{} \pi$, the parallel statement composed of ρ and π, can perform a τ action, denoting a successful communication.

Example. $\langle \textbf{new}\,(c); a; \textbf{new}\,(\bar{c}); b \rangle - a \rightarrow \langle c, \textbf{new}\,(\bar{c}); b \rangle - b \rightarrow \langle c, \bar{c}, E \rangle - \tau \rightarrow \langle E, E, E \rangle$.

Before we give the definition of the operational semantics of parallel statements, we introduce its semantic universe P.

DEFINITION 2.4 (Semantic universe P). Let A^∞ $(= A^* \cup A^\omega)$ denote the set of finite and infinite sequences or *words* of elements of A; let ε denote the empty word. We extend this set by allowing as the last element of a finite sequence a special element ∂, which denotes *deadlock*:

$$(w \in) A_\partial^\infty = A^* \cup A^* \cdot \{\partial\} \cup A^\omega.$$

Now we define $(p, q \in) P = \mathcal{P}_{nc}(A_\partial^\infty)$, the set of all non-empty and closed subsets of A_∂^∞. Let d denote the usual metric on A_∂^∞ (see the definition in I.1.1). We take $d_P = (d)_H$, the Hausdorff metric induced by d, as a metric on P. According to Proposition I.7, we have that (P, d_P) is a complete metric space.

DEFINITION 2.5 (Operational semantics $\mathcal{O}$). Let $\mathcal{O} = \text{Fixed Point } (\Phi)$, where $\Phi: (Par \to P) \to (Par \to P)$ is given, for $F \in Par \to P$, and $\rho \in Par$, by

$$\Phi(F)(\rho) = \begin{cases} \{\varepsilon\} & \text{if } \rho = \langle E, \cdots, E \rangle, \\ \{\partial\} & \text{if } \forall a \forall \rho' \quad [\rho - a \to \rho' \Rightarrow a \in C] \wedge \rho \neq \langle E, \cdots, E \rangle, \\ \bigcup \{a \cdot F(\rho') : \rho - a \to \rho' \wedge a \notin C\} & \text{otherwise.} \end{cases}$$

It is straightforward to show that Φ is a contraction and thus has a unique fixed point.

Note that an alternative equivalent definition of $\mathcal{O}$ could be given in terms of transition *sequences*, by putting, for instance, a word $a_1 \cdots a_n$ in $\mathcal{O}[\![\rho]\!]$ if and only if there exists a sequence

$$\rho - a_1 \to \rho_1 - a_2 \to \cdots - a_n \to \rho_n = \langle E, \cdots, E \rangle.$$

Since our language does not contain any constructs for recursion, we need not be able to describe infinite behavior. Therefore, it is not really necessary to define $\mathcal{O}$ using a contraction on a complete metric space. It would have been sufficient to take P as an ordinary set without any metric, and define $\mathcal{O}$ with an easy induction on the structure of statements. Our motivation for nevertheless exploiting metric structures here is given by the fact that in the next section we *will* deal with recursion and infinite behavior. There the use of some mathematical structure that can handle these, such as complete metric spaces, is obligatory. Our use of complete metric spaces at this stage can be seen as part of the introductory function of this section.

The operational semantics $\mathcal{O}$ can be best explained by giving a few examples.

Examples.

$$\mathcal{O}[\![\langle a \rangle]\!] = a \cdot \mathcal{O}[\![\langle E \rangle]\!] = a \cdot \{\varepsilon\} = \{a\},$$

$$\mathcal{O}[\![\langle \textbf{new}\,(a) \rangle]\!] = \{a\}, \qquad \mathcal{O}[\![\langle c \rangle]\!] = \{\partial\},$$

$$\mathcal{O}[\![\langle c, \bar{c} \rangle]\!] = \{\tau\}, \qquad \mathcal{O}[\![\langle a; b \rangle]\!] = a \cdot \mathcal{O}[\![\langle b \rangle]\!] = \{ab\},$$

$$\mathcal{O}[\![\langle \textbf{new}\,(a); b \rangle]\!] = \{a \cdot \mathcal{O}[\![\langle E, b \rangle]\!]\}, b \cdot \{\mathcal{O}[\![\langle a, E \rangle]\!]\} = \{ab, ba\}.$$

Note that a single communication $\langle c \rangle$, without a matching communication $\bar{c}$ in parallel, creates a deadlock.

Such an operational semantics is nice, because it is intuitively very clear. However, it is not *compositional* with respect to the binary syntactic operators ; and $\|$. For instance, there is no semantic operator $\tilde{;}: P \times P \to P$, corresponding to ;, such that for all s and t

$$\mathcal{O}[\![\langle s \,;\, t \rangle]\!] = \mathcal{O}[\![\langle s \rangle]\!] \,\tilde{;}\, \mathcal{O}[\![\langle t \rangle]\!].$$

This can be easily seen in the following argument. Suppose there *is* such an operator $\tilde{;}$. Then

$$\begin{aligned} \mathcal{O}[\![\langle \textbf{new}\,(a); b \rangle]\!] &= \mathcal{O}[\![\langle \textbf{new}\,(a) \rangle]\!] \,\tilde{;}\, \mathcal{O}[\![\langle b \rangle]\!] \\ &= [\text{since } \mathcal{O}[\![\langle \textbf{new}\,(a) \rangle]\!] = \mathcal{O}[\![\langle a \rangle]\!]] \\ &\quad \mathcal{O}[\![\langle a \rangle]\!] \,\tilde{;}\, \mathcal{O}[\![\langle b \rangle]\!] \\ &= \mathcal{O}[\![\langle a \,;\, b \rangle]\!], \end{aligned}$$

which yields a contradiction, as can be seen from the examples above.

The denotational semantics to be defined in a moment has the property that it is compositional with respect to the syntactic operators in L_T.

First, we define a suitable semantic universe.

DEFINITION 2.6 (Semantic universe $\bar{P}$). We define a complete metric space

$$(p, q \in)\bar{P} \text{ by } \bar{P} = \mathcal{P}_{nc}(A^\infty),$$

the set of nonempty and closed subsets of A^∞. Let d be the usual metric on A^∞; we define $d_P = (d)_H$.

The only difference between P and $\bar{P}$ is that the latter does not contain finite sequences ending in ∂.

DEFINITION 2.7 (Denotational semantics $\mathcal{D}$). Let $\mathcal{D}: L_T \to Cont \to \bar{P}$, where $Cont = \bar{P}$ denotes the set of *continuations*, be given by

$$\mathcal{D}[\![a]\!](p) = a \cdot p, \qquad \mathcal{D}[\![E]\!](p) = p,$$
$$\mathcal{D}[\![\textbf{new}\,(s)]\!](p) = p \parallel \mathcal{D}[\![s]\!](\{\varepsilon\}),$$
$$\mathcal{D}[\![s\ ;\ t]\!](p) = \mathcal{D}[\![s]\!](\mathcal{D}[\![t]\!](p)),$$

with $\parallel : P \times P \to P$ as defined below.

A continuation $p \in Cont$ denotes the semantics of the statement to be executed after the one to which $\mathcal{D}$ is applied. The meaning of a new-construct **new** (s) with continuation p is determined as follows. The meaning of s is computed with the empty continuation $\{\varepsilon\}$, which indicates that after s nothing remains to be done. Since s is to be executed in parallel with everything that follows, the result is composed in parallel with p, which indicates the remainder of the program after s.

DEFINITION 2.8 (Parallel composition $\parallel$). Let $\parallel : P \times P \to P$ be such that it satisfies, for $p, q \in P$,

$$p \parallel q = p \mathbin{\lfloor\!\lfloor} q \cup q \mathbin{\lfloor\!\lfloor} p \cup p \mid q,$$

where

$$p \mathbin{\lfloor\!\lfloor} q = \bigcup \{a \cdot (p_a \parallel q) \colon p_a \neq \varnothing\} \cup \{q \colon \varepsilon \in p\},$$
$$p \mid q = \bigcup \{\tau \cdot (p_c \parallel q_{\bar{c}}) \colon p_c \neq \varnothing \neq q_{\bar{c}}\},$$

with $p_a = \{w \colon a \cdot w \in p\}$, the set containing all the postfixes of a in p.

The above definition is self-referential and needs some justification. Formally, we can define $\parallel$ as the fixed point of a contraction $\Psi : (\bar{P} \times \bar{P} \to \bar{P}) \to (\bar{P} \times \bar{P} \to \bar{P})$ given, for $f \in \bar{P} \times \bar{P} \to \bar{P}$, by

$$\Psi(f)(p, q) = p \mathbin{\lfloor\!\lfloor}_f q \cup q \mathbin{\lfloor\!\lfloor}_f p \cup p \mid_f q,$$

where

$$p \mathbin{\lfloor\!\lfloor}_f q = \bigcup \{a \cdot f(p_a, q) \colon p_a \neq \varnothing\} \cup \{q \colon \varepsilon \in p\},$$
$$p \mid_f q = \bigcup \{\tau \cdot (f(p_c, q_{\bar{c}})) \colon p_c \neq \varnothing \neq q_{\bar{c}}\}.$$

Note that $\mathcal{D}$ is compositional with respect to ";". The corresponding semantic operator $;: ((\bar{P} \to \bar{P}) \times (\bar{P} \to \bar{P})) \to (\bar{P} \to \bar{P})$ is not expressed explicitly in the definition of $\mathcal{D}$. For completeness sake, we give its definition. We have, for $f, g \in \bar{P} \to \bar{P}$,

$$f \tilde{;} g = \lambda p \cdot f(g(p)).$$

2.1. Semantic equivalence of $\mathcal{O}$ and $\mathcal{D}$. After having defined $\mathcal{O}$ and $\mathcal{D}$ for *Par* and L_T, we next discuss the relationship between the two semantics. We shall compare $\mathcal{O}$ and $\mathcal{D}$ by relating both to an intermediate semantics $\mathcal{O}' : Par \to P$, given in the following definition.

DEFINITION 2.9 (Intermediate semantics $\mathcal{O}'$). Let $\mathcal{O}' = \text{Fixed point } (\Phi')$, where $\Phi' : (Par \to \bar{P}) \to (Par \to \bar{P})$ is given, for $F \in Par \to \bar{P}$ and $\rho \in Par$, by

$$\Phi'(F)(\rho) = \begin{cases} \{\varepsilon\} & \text{if } \rho = \langle E, \cdots, E \rangle, \\ \bigcup \{a \cdot F(\rho') : \rho - a \to \rho'\} & \text{otherwise.} \end{cases}$$

Note that in Φ', as opposed to Φ, single-sided communication steps $a \in C$ are allowed. The difference between $\mathcal{O}$ and $\mathcal{O}'$ can be illustrated by giving a few examples:

$$\mathcal{O}[\![\langle c \rangle]\!] = \{\partial\}, \qquad \mathcal{O}[\![\langle c, \bar{c} \rangle]\!] = \{\tau\},$$

$$\mathcal{O}'[\![\langle c \rangle]\!] = \{c\}, \qquad \mathcal{O}'[\![\langle c, \bar{c} \rangle]\!] = \{c\bar{c}, \bar{c}c, \tau\}.$$

The relationship between $\mathcal{O}$ and $\mathcal{O}'$ will be expressed using the following abstraction operation.

DEFINITION 2.10 (Abstraction operator α). We define an abstraction operator $\alpha : \bar{P} \to P$ by

$$\alpha(p) = \begin{cases} \{\partial\} & \text{if } \forall a [p_a \neq \varnothing \Rightarrow a \in C], \\ \bigcup \{a \cdot (\alpha(p_a)) : a \notin C \wedge p_a \neq \varnothing\} \cup \{\varepsilon : \varepsilon \in p\} & \text{otherwise,} \end{cases}$$

with p_a as in Definition 2.8. (For a justification of this self-referential definition see the remark following Definition 2.8.)

The definition of α can be understood as follows. If all the words $w \in p$ begin with a communication action $a \in C$, we have operationally a deadlock, since no single communication action is allowed. Therefore, we then have that $\alpha(p) = \{\partial\}$. In the last case, $\alpha(p)$ contains all the words in p that begin with a noncommunication action $a \in A \backslash C$, with α recursively applied to p_a, the set of postfixes of a; additionally, $\alpha(p)$ contains ε if $\varepsilon \in p$.

The following theorem can be proved straightforwardly.

THEOREM 2.11. *For all* $F \in Par \to \bar{P}[\Phi(\alpha \circ F) = \alpha \circ \Phi'(F)]$.

Since Φ and Φ' are contractions and thus have unique fixed points, it follows that we have Corollary 2.12.

COROLLARY 2.12. $\mathcal{O} = \alpha \circ \mathcal{O}'$.

Proof. We have that $\alpha \circ \mathcal{O}' = \alpha \circ \Phi'(\mathcal{O}') = \Phi(\alpha \circ \mathcal{O}')$. Thus both $\alpha \circ \mathcal{O}'$ and $\mathcal{O}$ are fixed points of Φ, which implies that they are equal.

The relationship between $\mathcal{O}'$ and $\mathcal{D}$ can be elegantly expressed using the following mapping.

DEFINITION 2.13. We define $\sim : (L_T \to Cont \to \bar{P}) \to (Par \to \bar{P})$ as follows. We denote, for $F \in L_T \to Cont \to \bar{P}$, $\sim(F)$ by $\tilde{F}$ and put

$$\tilde{F} = \lambda \rho \in Par \cdot (F(s_1)(\{\varepsilon\}) \parallel \cdots \parallel F(s_n)(\{\varepsilon\})),$$

with $\rho = \langle s_1, \cdots, s_n \rangle$.

A simple consequence, using the associativity of $\parallel$, of this definition is $\tilde{F}(\rho^\wedge \tau) = \tilde{F}(\rho) \parallel \tilde{F}(\tau)$. If the function $\tilde{F}$ takes a parallel statement $\langle s_1, \cdots, s_n \rangle$ as an argument, then the F values of all the substatements s_i supplied with the empty continuation $\{\varepsilon\}$ are computed and next composed in parallel.

Now we can prove that $\mathcal{O}' = \tilde{\mathcal{D}}$. It is a corollary of the following theorem.

THEOREM 2.14. $\Phi'(\tilde{\mathcal{D}}) = \tilde{\mathcal{D}}$.

Proof. The proof uses induction on the structure of parallel statements. We treat one typical case, leaving the other ones to the reader. Consider $\rho^\wedge \pi \in Par$ and suppose

$\rho \neq \langle E, \cdots, E\rangle$ and $\pi \neq \langle E, \cdots, E\rangle$. Suppose we already know that $\Phi'(\tilde{\mathscr{D}})(\rho) = \tilde{\mathscr{D}}(\rho)$ and $\Phi'(\mathscr{D})(\pi) = \tilde{\mathscr{D}}(\pi)$. We show that $\Phi'(\tilde{\mathscr{D}})(\rho^\wedge\pi) = \tilde{\mathscr{D}}(\rho^\wedge\pi)$.

$$
\begin{aligned}
\Phi'(\tilde{\mathscr{D}})(\rho^\wedge\pi) &= \bigcup\{a \cdot \tilde{\mathscr{D}}(\rho') \colon \rho^\wedge\pi - a \to \rho'\} \\
&= [\text{definition of} \to (2.3(5) \text{ and } (6))] \\
&\quad \bigcup\{a \cdot \tilde{\mathscr{D}}(\rho'^\wedge\pi) \colon \rho - a \to \rho'\} \cup \bigcup\{a \cdot \tilde{\mathscr{D}}(\rho^\wedge\pi') \colon \pi - a \to \pi'\} \\
&\quad \cup \bigcup\{\tau \cdot \tilde{\mathscr{D}}(\rho'^\wedge\pi') \colon \rho - c \to \rho' \wedge \pi - \bar{c} \to \pi'\} \\
&= [\text{definition} \sim] \\
&\quad \bigcup\{a \cdot (\tilde{\mathscr{D}}(\rho') \parallel \tilde{\mathscr{D}}(\pi)) \colon \rho - a \to \rho'\} \\
&\quad \cup \bigcup\{a \cdot (\tilde{\mathscr{D}}(\rho) \parallel \tilde{\mathscr{D}}(\pi')) \colon \pi - a \to \pi'\} \\
&\quad \cup \bigcup\{\tau \cdot (\tilde{\mathscr{D}}(\rho') \parallel \tilde{\mathscr{D}}(\pi')) \colon \rho - c \to \rho' \wedge \pi - \bar{c} \to \pi'\} \\
&= [\text{definitions } \lfloor\!\lfloor \text{ and } |] \\
&\quad (\bigcup\{a \cdot \tilde{\mathscr{D}}(\rho') \colon \rho - a \to \rho'\} \lfloor\!\lfloor\, \tilde{\mathscr{D}}(\pi)) \\
&\quad \cup (\bigcup\{a \cdot \tilde{\mathscr{D}}(\pi') \colon \pi - a \to \pi'\} \lfloor\!\lfloor\, \tilde{\mathscr{D}}(\rho)) \\
&\quad \cup (\bigcup\{c \cdot \tilde{\mathscr{D}}(\rho') \colon \rho - c \to \rho'\} \mid \bigcup\{\bar{c} \cdot \tilde{\mathscr{D}}(\pi') \colon \pi - \bar{c} \to \pi'\}) \\
&= (\Phi'(\tilde{\mathscr{D}})(\rho) \lfloor\!\lfloor\, \tilde{\mathscr{D}}(\pi)) \cup (\Phi'(\tilde{\mathscr{D}})(\pi) \lfloor\!\lfloor\, \tilde{\mathscr{D}}(\rho)) \\
&\quad \cup (\Phi'(\tilde{\mathscr{D}})(\rho) \mid \Phi'(\tilde{\mathscr{D}})(\pi)) \\
&= [\text{induction}] \\
&\quad (\tilde{\mathscr{D}}(\rho) \lfloor\!\lfloor\, \tilde{\mathscr{D}}(\pi)) \cup (\tilde{\mathscr{D}}(\pi) \lfloor\!\lfloor\, \tilde{\mathscr{D}}(\rho) \cup (\tilde{\mathscr{D}}(\rho) \mid \tilde{\mathscr{D}}(\pi))) \\
&= \tilde{\mathscr{D}}(\rho) \parallel \tilde{\mathscr{D}}(\pi) \\
&= \tilde{\mathscr{D}}(\rho^\wedge\pi).
\end{aligned}
$$

□

COROLLARY 2.15. $\mathscr{O}' = \tilde{\mathscr{D}}$.

Combining Corollaries 2.12 and 2.15 now yields the main theorem of this section.

MAIN THEOREM 2.16. $\mathscr{O} = \alpha \circ \tilde{\mathscr{D}}$.

COROLLARY 2.17. *For all* $s \in L_T[\mathscr{O}[\![\langle s\rangle]\!] = \alpha(\mathscr{D}[\![s]\!](\{\varepsilon\}))]$.

3. The language POOL. In this paper, we compare different semantic models of a language that we call POOL: Parallel Object-Oriented Language. It is a simplified version of a language called POOL-T, which is defined in [Am85]. (For an account of the design considerations for POOL-T, see [Am86] and [Am87].) The simplification is twofold. First, we omitted certain language constructs from POOL-T (such as the select statement and the method call) as well as some of the protection mechanisms offered by the definition of classes (such as different classes having different instances of variables and method definitions). We have done this in order to make life somewhat easier: the semantic definitions are shorter and so are the proofs of the theorems. We feel justified in doing so, since it is straightforward to extend the approach of this paper to the full language. Second, we give an abstract syntactic description of POOL, which is a simplified version of the formal description of POOL-T.

A POOL program describes the behavior of a whole system in terms of its constituents, *objects*. Objects contain some internal data, and some procedures that act on these data (these are called *methods* in the object-oriented jargon). Objects are entities of a dynamic nature: they can be created dynamically, their internal data can be modified, and they have an internal activity of their own. At the same time they are units of protection: the internal data of one object are not directly accessible for other objects.

An object uses *variables* (more specifically: instance variables) to store its internal data. Each variable can contain the *name* of an object (another object, or, possibly, the object under consideration itself). An assignment to a variable can make it refer to an object different from the object referred to before. The variables of one object cannot be accessed directly by other objects. They can only be read and changed by the object itself.

Objects can interact by sending *messages* to each other. A message is a request for the receiver to execute a certain method. Messages are sent and received explicitly. In sending a message, the sender mentions the destination object, the method to be executed, and possibly a parameter (which is again an object name) to be passed to this method. After this, its activity is suspended. The receiver can specify the set of methods that will be accepted, but it can place no restrictions on the identity of the sender or on the parameters of messages. If a message arrives specifying an appropriate method, the method is executed with the parameters contained in the message. Upon termination, this method delivers a result (an object name), which is returned to the sender of the message. The latter then resumes its own execution. Note that this form of communication strongly resembles the rendez-vous mechanism of Ada [ANSI83].

A method can accesss the variables of the object by which it is executed (the receiver of a message). Furthermore, it has a formal parameter, which is initialized to the actual parameter specified in the message.

When an object is created, a local activity is started: the object's *body*. When several objects have been created, their bodies execute in parallel. This is the way parallelism is introduced into the language. Synchronization and communication takes place by sending messages, as described above.

Objects are grouped into *classes*. All objects in one class (the *instances* of that class) execute the same body. In creating an object, only its desired class must be specified. In this way a class serves as a blueprint for the creation of its instances.

At this point, it might be useful to emphasize the distinction between an object and its name. Objects are intuitive entities as described above. In this paper, there will appear no mathematical construction that directly models a single object with all its dynamic properties (although it would be interesting to see a semantics that does this). Object names, on the other hand, are modeled explicitly as elements of some abstract set *Obj*. Object names are only *references* to objects. On its own, an object name gives little information about the object it refers to. In fact, object names are just sufficient to distinguish the individual objects from each other. Note that variables and parameters contain object names, and that expressions result in object names, not objects. If in the sequel we speak, for example, of "the object α," we hope the reader will understand that the object with name α is meant.

Now we describe the (abstract) syntax of the language POOL. We assume that the following sets of syntactic elements are given:

$(x \in) IVar$ (instance variables),

$(u \in) TVar$ (temporary variables),

$(C \in) CName$ (class names),

$(m \in) MName$ (method names).

DEFINITION 3.1 (Expressions, statements, units). We define the set of expressions $(e \in) L_E$ and the set of statements $(s \in) L_S$ by

$$e ::= x \mid u \mid e_1!m(e_2) \mid \textbf{new}(C) \mid s\ ;\ e \mid \textbf{self},$$

$$s ::= x \leftarrow e \mid u \leftarrow e \mid \textbf{answer } m \mid s_1 ; s_2 \mid \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi} \mid \textbf{do } e \textbf{ then } s \textbf{ od}.$$

The set $(U \in) \mathit{Unit}$ of units is defined by

$$U ::= \langle (C_1 \Leftarrow s_1, \cdots, C_n \Leftarrow s_n), \quad (m_1 \Leftarrow \langle u_1, e_1 \rangle, \cdots, m_k \Leftarrow \langle u_k, e_k \rangle) \rangle.$$

We write $C \Leftarrow s \in U$ if there exists an i such that $C_i = C$ and $s_i = s$. Similarly, we write $m \Leftarrow \langle u, e \rangle \in U$.

An instance variable or a temporary variable used as an expression will yield as its value the object name that is currently stored in that variable.

The next kind of expression is a send expression. Here, e_1 is the destination object, to which the message will be sent, m is the method to be invoked, and e_2 is the parameter. When a send expression is evaluated, the destination expression and the parameter expression are evaluated successively. Next, the message is sent to the destination object. When this object answers the message, the corresponding method is executed, that is, the formal parameter is initialized to the name of the object in the message, and the expression in the method definition is evaluated. The value that results from this evaluation is sent back to the sender of the message and this will be the value of the send expression.

A new-expression indicates that a new object is to be created, an instance of the indicated class. Its body starts executing in parallel with all other objects in the system. The result of the new-expression is (the name of) this newly created object.

An expression may also be preceded by a statement. In this case the statement is executed before the expression is evaluated.

The expression **self** always results in the name of the object that is executing this expression.

The first two kinds of statements are assignments, to an instance variable and to a temporary variable, respectively. An assignment is executed by first evaluating the expression on the right, and then making the variable on the left refer to the resulting object.

An answer statement indicates that a message is to be answered. The object executing the answer statement waits until a message arrives with a method name that is specified by the answer statement. Then it executes the method (after initializing the formal parameter). The result of the method is sent back to the sender of the message, and the answer statement terminates.

Sequential composition, conditionals, and loops have the usual meaning.

Units are the programs of POOL. A unit consists of a number of definitions of class bodies and methods. If a unit is to be executed, a single new instance of the *last* class defined in the unit is created and execution of its body is started. This object has the task to start the whole system, by creating new objects and putting them to work.

The relationship between POOL and POOL-T is the following: POOL is obtained from POOL-T via two successive simplifications. First, certain language constructs from POOL-T are omitted (such as the select statement) as well as some of the protection mechanisms in POOL-T, which are offered by the definition of classes (such as different classes having different variables and method definitions). Second, some syntactical simplifications are performed and some context information is omitted (POOL-T is a statically typed language, whereas POOL is not). The reason for making the first simplification is simply lack of space, to which should be added the consideration that it would be straightforward to extend our results to the full language. The sole reason for making the second simplification is that POOL-T is a practical programming language, for which readability, among others, is more important than syntactic

simplicity. Therefore, it is convenient to take a simplified language, POOL, as the semantic core of POOL-T.

If one compares the version of POOL described in this paper with the one given in [ABKR86(a)] and [ABKR86(b)], some minor differences can be observed. (For example, in the send expression of Definition 3.1 above only one parameter can be specified whereas in the definitions of the papers mentioned an arbitrary number of parameters is allowed.) However, it can easily be seen that it is straightforward to adapt the definitions and proofs given in this paper such that they apply to the version of POOL occurring in [ABKR86(a)] and [ABKR86(b)].

4. An operational semantics for POOL. In this section we give the definition of an operational semantics for POOL, which is a modified version of the one given in [ABKR86(a)]. (At the end of this section, we shall compare both models in some detail.) It is based on a *transition relation* and will be defined as the fixed point of a suitable contraction. For this purpose, we introduce a number of syntactic and semantic notions.

First of all, we introduce the set of objects.

DEFINITION 4.1 (Objects). We assume given a set $AObj$ of names for *active* objects together with a function

$$\nu : \mathcal{P}_{fin}(AObj) \to AObj$$

such that $\nu(X) \notin X$, for every finite $X \subseteq AObj$. Given a set X of object names, the function ν yields a new name not in X.

Further, we define

$$Obj = AObj \cup SObj,$$

where $SObj$ is the set of so-called *standard* objects, to be introduced in Appendix III.

A possible example of such a set $AObj$ and function ν could be obtained by setting

$$AObj = \mathbb{N}, \qquad \nu(X) = \max\{n : n \in X\} + 1.$$

In POOL, a few standard classes, the instances of which are called standard objects, are predefined; examples are the classes of Booleans and integers. The semantic treatment of these standard objects is somewhat different from the way the active objects (which are created during the execution of a POOL program) are treated. Because we want to formulate our semantic models as concisely as possible in order to focus on the correctness proof, the standard objects are treated in Appendix III.

Next, it is convenient to extend the sets L_E of expressions and L_S of statements by adding some auxiliary syntactic constructs.

DEFINITION 4.2 ($L_{E'}$, $L_{S'}$). Let $(e \in) L_{E'}$ and $(s \in) L_{S'}$ be defined by

$$e ::= x \mid u \mid e_1 ! m(e_2) \mid \textbf{new}(C) \mid s\,;\, e \mid \textbf{self} \mid \alpha \mid (e, \phi),$$

$$s ::= x \leftarrow e \mid u \leftarrow e \mid \textbf{answer}\ m \mid s_1\,;\, s_2 \mid \textbf{if}\ e\ \textbf{then}\ s_1\ \textbf{else}\ s_2\ \textbf{fi} \mid \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od}$$
$$\mid \textbf{release}(\beta, s) \mid (e, \psi)$$

with $\alpha, \beta \in AObj$, $\phi \in L_{PE}$, and $\psi \in L_{PS}$. Here the sets of *parameterized expressions* $(\phi \in) L_{PE}$ and *parameterized statements* $(\psi \in) L_{PS}$ are given by

$$\phi ::= \lambda u \cdot e, \qquad \psi ::= \lambda u \cdot s,$$

with the restriction that u does not occur at the left-hand side of an assignment in e or s. For $\alpha \in Obj$, $\phi = \lambda u \cdot e$, and $\psi = \lambda u \cdot s$, we shall use $\phi(\alpha)$ and $\psi(\alpha)$ to denote the expression and the statement obtained by syntactically substituting α for all free

occurrences of u in ϕ and ψ, respectively. The restriction just mentioned ensures that the result of this substitution again is a well-formed expression or statement.

Let us explain the new syntactic constructs. In addition to what we already had in L_E, an expression $e \in L_{E'}$ can be an *active* object α or a pair (e, ϕ) of an expression e and a parameterized expression ϕ. The latter will be executed as follows: First the expression e is evaluated, then the result β is substituted in ϕ and $\phi(\beta)$ is executed. As new statements we have release statements **release** (β, s) and parameterized statements (e, ϕ). If the statement **release** (β, s) is executed, the active object β will start executing the statement s (in parallel to the objects that are already executing). The release statement will be used in the description of the communication between two objects (see Definition 4.8 below). The interpretation of (e, ψ) is similar to that of (e, ϕ).

DEFINITION 4.3 (Empty statement). The set $L_{S'}$, as given in the definition above, is extended with a special element E, denoting the *empty statement.* This extended set is again called $L_{S'}$. Note that we do *not* have elements like s ; E or **do** e **then** E **od** in $L_{S'}$. (There is, however, one exception: we *do* allow E in **if** e **then** s **else** E **fi**, which is needed in Definition 4.8(A7) below.)

DEFINITION 4.4 (States). The set of states $(\sigma \in)\Sigma$ is defined by

$$\Sigma = (AObj \to IVar \to Obj) \times (AObj \to TVar \to Obj) \times \mathcal{P}_{fin}(AObj).$$

The three components of σ are denoted by $\langle \sigma_1, \sigma_2, \sigma_3 \rangle$. The first and the second component of a state store the values of the instance variables and the temporary variables of each active object. The third component contains the object names currently in use. We need it in order to give unique names to newly created objects.

We shall use the following variant notation. By $\sigma\{\beta/\alpha, x\}$ (with $x \in IVar$) we shall denote the state σ' that is as σ but for the value of $\sigma_1'(\alpha)(x)$, which is β. Similarly, we denote by $\sigma\{\beta/\alpha, u\}$ (with $u \in TVar$) the state σ' that is as σ but for the value of $\sigma_1'(\alpha)(u)$, which is β.

DEFINITION 4.5 (Labelled statements). The set of *labelled statements* $((\alpha, s) \in)LStat$ is given by

$$LStat = AObj \times L_{S'}.$$

A labelled statement (α, s) should be interpreted as a statement s that will be executed by the active object α.

Sometimes, we also need labelled parameterized statements. Therefore, we extend $LStat$:

$$LStat' = LStat \cup (AObj \times L_{PS}).$$

A pair (α, ψ) indicates that the active object α will execute the statement ψ as soon as it receives a value that it can supply to ψ as an argument.

Before we can give the definition of a transition relation for POOL, we first have to explain which *configurations* and *transition labels* we are going to use.

DEFINITION 4.6 (Configurations). The set of configurations $(\rho \in)Conf$ is given by

$$Conf = \mathcal{P}_{fin}(LStat) \times \Sigma.$$

We also introduce

$$Conf' = \mathcal{P}_{fin}(LStat') \times \Sigma.$$

Typical elements of $Conf$ and $Conf'$ will also be indicated by $\langle X, \sigma \rangle$ and $\langle Y, \sigma \rangle$.

We shall consider only configurations $\langle X, \sigma\rangle$ that are *consistent* in the following sense. For $X = \{(\alpha_1, s_1), \cdots, (\alpha_k, s_k)\}$, we call $\langle X, \sigma\rangle$ consistent if the following conditions are satisfied:

$$\forall i, j \in \{1, \cdots, k\} \quad [i \neq j \Rightarrow \alpha_i \neq \alpha_j] \quad \text{and} \quad \{\alpha_1, \cdots, \alpha_k\} \subseteq \sigma_3.$$

Whenever we introduce a configuration $\langle X, \sigma\rangle$, it will be tacitly assumed that it is consistent.

A configuration $\langle X, \sigma\rangle$, consisting of a finite set X of labelled statements and a state σ, represents a "snapshot" of the execution of a POOL program. It shows what objects are active and what statements they are executing; furthermore, it contains a state σ, in which the values of the variables of the active objects as well as the set of object names currently in use are stored.

DEFINITION 4.7 (Transition labels). The set of *transition labels* $(\lambda \in)\Lambda$ is given by

$$\Lambda = \{\tau\} \cup \{(\alpha, \beta_1 ! m(\beta_2)) : \alpha, \beta_1 \in AObj, \beta_2 \in Obj\} \cup \{(\beta ? m) : \beta \in AObj\}.$$

These labels will be used in the definition of the transition relation below and are to be interpreted as follows. The label τ indicates a so-called *computation* step. Next, $(\alpha, \beta_1 ! m(\beta_2))$ indicates that object α sends a message to object β_1 requesting the execution of the method m with parameter β_2. Finally, $(\beta ? m)$ indicates that the object β is willing to answer a message specifying the method m.

Now we are ready to define a transition relation for POOL.

DEFINITION 4.8 (Transition relation). Let $U \in Unit$. We define a *labelled transition relation*

$$-U\rightarrow \subseteq Conf \times \Lambda \times Conf'.$$

Triples $\langle \rho_1, \lambda, \rho_2\rangle \in -U\rightarrow$ will be called *transitions* and are denoted by

$$\rho_1 - U, \lambda \rightarrow \rho_2.$$

Such a transition reflects a possible execution step of type λ of the configuration ρ_1, yielding a new configuration ρ_2. The relation $-U\rightarrow$ is defined as the smallest relation satisfying the following properties.

AXIOMS:

(A1) $\langle\{(\alpha, (x, \psi))\}, \sigma\rangle - U, \tau \rightarrow \langle\{(\alpha, (\sigma_1(\alpha)(x), \psi))\}, \sigma\rangle$.

(A2) $\langle\{(\alpha, (u, \psi))\}, \sigma\rangle - U, \tau \rightarrow \langle\{(\alpha, (\sigma_2(\alpha)(u), \psi))\}, \sigma\rangle$.

(A3) $\langle\{(\alpha, (\beta_1 ! m(\beta_2), \psi))\}, \sigma\rangle - U, (\alpha, (\beta_1 ! m(\beta_2))) \rightarrow \langle\{(\alpha, \psi)\}, \sigma\rangle$.

(A4) $\langle\{(\alpha, (\textbf{new}\,(C), \psi))\}, \sigma\rangle - U, \tau \rightarrow \langle\{(\alpha, (\beta, \psi)), (\beta, s_C)\}, \sigma'\rangle$, where $C \Leftarrow s_C \in U, \beta = \nu(\sigma_3), \sigma' = \langle \sigma_1, \sigma_2, \sigma_3 \cup \{\beta\}\rangle$.

(A5) $\langle\{(\alpha, z \leftarrow \beta)\}, \sigma\rangle - U, \tau \rightarrow \langle\{(\alpha, E)\}, \sigma\{\beta/\alpha, z\}\rangle$ for $z \in IVar \cup TVar$.

(A6) $\langle\{(\alpha, \textbf{answer}\ m)\}, \sigma\rangle - U, (\alpha ? m) \rightarrow \langle\{(\alpha, E)\}, \sigma\rangle$.

(A7) $\langle\{(\alpha, \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od})\}, \sigma\rangle - U, \tau \rightarrow$
$\langle\{(\alpha, \textbf{if}\ e\ \textbf{then}\ (s; \textbf{do}\ e\ \textbf{then}\ s\ \textbf{od})\ \textbf{else}\ E\ \textbf{fi})\}, \sigma\rangle$.

RULES:

(R1) If $\langle\{(\alpha, (e, \lambda u \cdot z \leftarrow u))\}, \sigma\rangle - U, \lambda \rightarrow \rho$,
then $\langle\{(\alpha, z \leftarrow e)\}, \sigma\rangle - U, \lambda \rightarrow \rho$, for $a \in IVar \cup TVar$.

(R2) If $\langle\{(\alpha, s)\}, \sigma\rangle - U, \lambda \rightarrow \langle\{(\alpha, s')\} \cup X, \sigma'\rangle$,

then $\langle\{(\alpha, s\,;\, t)\}, \sigma\rangle - U, \lambda \to \langle\{(\alpha, s'\,;\, t)\} \cup X, \sigma'\rangle$
(read t instead of $s'\,;\, t$ if $s' = E$).
if $\langle\{(\alpha, s)\}, \sigma\rangle - U, \lambda \to \langle\{(\alpha, \psi)\} \cup X, \sigma'\rangle$,
then $\langle\{(\alpha, s\,;\, t)\}, \sigma\rangle - U, \lambda \to \langle\{(\alpha, \lambda u \cdot (\psi(u)\,;\, t))\} \cup X, \sigma'\rangle$.

(R3) If $\langle\{(\alpha, s_i)\}, \sigma\rangle - U, \lambda \to \rho$, then $\langle\{(\alpha, \textbf{if } \beta \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi})\}, \sigma\rangle - U, \lambda \to \rho$,
where
$$s_i = \begin{cases} s_1 & \text{if } \beta = tt, \\ s_2 & \text{if } \beta = ff. \end{cases}$$

(R4) If $\langle\{(\alpha, t), (\beta, s)\}, \sigma\rangle - U, \lambda \to \rho$, then $\langle\{(\alpha, \textbf{release } (\beta, s)\,;\, t\}, \sigma\rangle - U, \lambda \to \rho$
(read **release** (β, s) instead of **release** $(\beta, s)\,;\, t$ if $t = E$).

(R5) If $\langle\{(\alpha, (e, \lambda u \cdot \textbf{if } u \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi}))\}, \sigma\rangle - U, \lambda \to \rho$,
then $\langle\{(\alpha, \textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi})\}, \sigma\rangle - U, \lambda \to \rho$.
(Here s_2 is allowed to be E.)

(R6) If $\langle\{(\alpha, ((e_1, \lambda u_1 \cdot (e_2, \lambda u_2 \cdot u_1!m(u_2))), \psi))\}, \sigma\rangle - U, \lambda \to \rho$,
then $\langle\{(\alpha, (e_1!m(e_2), \psi))\}, \sigma\rangle - U, \lambda \to \rho$.

(R7) If $\langle\{(\alpha, s\,;\, (e, \psi))\}, \sigma\rangle - U, \lambda \to \rho$, then $\langle\{(\alpha, (s\,;\, e, \psi))\}, \sigma\rangle - U, \lambda \to \rho$.

(R8) If $\langle\{(\alpha, (e, \lambda u \cdot (\phi(u), \psi)))\}, \sigma\rangle - U, \lambda \to \rho$,
then $\langle\{(\alpha, ((e, \phi), \psi))\}, \sigma\rangle - U, \lambda \to \rho$.

(R9) If $\langle\{(\alpha, \psi(\beta))\}, \sigma\rangle - U, \lambda \to \rho$, then $\langle\{(\alpha, (\beta, \psi))\}, \sigma\rangle - U, \lambda \to \rho$, for $\beta \in Obj$.
If $\langle\{(\alpha, \psi(\alpha))\}, \sigma\rangle - U, \lambda \to \rho$, then $\langle\{(\alpha, (\textbf{self}, \psi))\}, \sigma\rangle - U, \lambda \to \rho$.

(R10) If $\langle X, \sigma\rangle - U, \lambda \to \langle X', \sigma'\rangle$, then $\langle X \cup Y, \sigma\rangle - U, \lambda \to \langle X' \cup Y, \sigma'\rangle$.

(R11) If $\langle X, \sigma\rangle - U, (\alpha, \beta_1!m(\beta_2)) \to \langle\{(\alpha, \psi)\} \cup X', \sigma\rangle$ and
$\langle Y, \sigma\rangle - U, \beta_1?m \to \langle\{(\beta_1, s)\} \cup Y', \sigma\rangle$,
then $\langle X \cup Y, \sigma\rangle - U, \tau \to$
$\langle\{(\beta_1, (e_m, \lambda u \cdot (u_m \leftarrow \sigma_2(\beta_1)(u_m); \textbf{release } (\alpha, \psi(u))\,;\, s)))\} \cup X' \cup Y', \sigma'\rangle$,
where $\sigma' = \sigma\{\beta_2/\beta_1, u_m\}$, and $m \Leftarrow \langle u_m, e_m\rangle \in U$.

(End of definition.)

The general scheme for the evaluation of an expression is very similar to the approach taken in [AB88]. Expressions always occur in the context of a (possibly parameterized) statement, such as $x \leftarrow e$. A statement containing e as a subexpression is transformed into a pair (e, ψ) of the expression e and a parameterized statement ψ by application of one of the rules. (In our example, $x \leftarrow e$ becomes $(x, \lambda u \cdot x \leftarrow u)$ by an application of (R1).) Then e is evaluated, using the axioms and rules, and results in some value $\beta' \in Obj$. (Applying (A1) transforms $(x, \lambda u \cdot x \leftarrow u)$ of our example into $(\beta', \lambda u \cdot x \leftarrow u)$, for some $\beta' \in Obj$.) Next, an application of (R9) will put the resulting object β' back into the original context ψ (yielding $x \leftarrow \beta'$ in our example). Finally, the statement $\psi(\beta')$ is further evaluated by using the axioms and the rules. (The evaluation of $x \leftarrow \beta'$ results, by using (A6), in a transformation of the state.)

Let us briefly explain some of the axioms and rules above.

In (A4) a new object is created. Its name β is obtained by applying the function ν to the set σ_3 of the active object names currently in use and is delivered as the result of the evaluation of **new** (C). The body s_C of class C, defined in the unit U, is going to be evaluated by β. Note that the state σ is changed by extending σ_3 with β.

In (R8), the evaluation of an expression pair (e, ϕ), where ϕ is a parameterized expression, in the context of a parameterized statement ψ is reduced to the evaluation of the expression e in the context of the adapted parameterized statement $\lambda u \cdot (\phi(u), \psi)$.

Rule (R11) describes the communication rendez-vous of POOL. If the object α is sending a message to object β_1, requesting the execution of the method m and if the object β_1 is willing to answer such a message, then the following happens: The object β_1 starts executing the expression e_m, which corresponds to the definition of the method m in U, while its state $\sigma_2(\beta_1)$ is changed by setting u_m, the formal parameter belonging to m, to β_2, the parameter sent by the object α to β_1. After the execution of e_m, the object β_1 continues by executing $u_m \leftarrow \sigma_2(\beta_1)(u_m)$, which restores the old value of u_m, followed by the statement **release** $(\alpha, \psi(u))$; s. The execution of **release** $(\alpha, \psi(u))$ will reactivate the object α, which starts executing $\psi(u)$, the statement obtained by substituting the result u of the execution of e_m into ψ. Note that during the execution of e_m the object α is nonactive, as can be seen from the fact that α does not occur as the name of any labelled statement in the configuration resulting from this transition. Finally, the object β_1 proceeds with the execution of the statement s, which is the remainder of its body.

(Note that we have not incorporated any transitions for the standard objects; this is done in Appendix III.)

Now we are ready for the definition of the operational semantics of POOL. It will use the following semantic universe.

DEFINITION 4.9 (Semantic universe P). Let $(w \in)\Sigma_\partial^\infty = \Sigma^* \cup \Sigma^\omega \cup \Sigma^* \cdot \{\partial\}$, the set of *streams*. We define

$$(p, q \in)P = \Sigma \to \mathcal{P}_{ncompact}(\Sigma_\partial^\infty),$$

where $\mathcal{P}_{ncompact}(\Sigma_\partial^\infty)$ is the set of all nonempty compact subsets of Σ_∂^∞, and the symbol ∂ denotes deadlock. The set P is a complete metric space when supplied with the usual metric (see Definition I.6).

The elements of P will be used to represent the operational meanings of statements and units. For a given state $\sigma \in \Sigma$, the set $p(\sigma)$ contains streams $w \in \Sigma_\partial^\infty$, which are sequences of states representing possible computations. They can be of one of three forms. If $w \in \Sigma^*$, it stands for a finite normally terminating computation. If $w \in \Sigma^\omega$, it represents an infinite computation. Finally, if $w \in \Sigma^* \cdot \{\partial\}$, it reflects a finite abnormally terminating computation, which is indicated by the symbol ∂ for deadlock.

DEFINITION 4.10 (Operational semantics for POOL). We define the operational semantics of finite subsets of labelled statements. Let, for a unit $U \in \mathit{Unit}$, the function

$$\Phi_U : (\mathcal{P}_{fin}(\mathit{LStat}) \to P) \to (\mathcal{P}_{fin}(\mathit{LStat}) \to P)$$

be given, for $F \in \mathcal{P}_{fin}(\mathit{LStat}) \to P$ and $X \in \mathcal{P}_{fin}(\mathit{LStat})$ by

$$\Phi_U(F)(X) = \lambda\sigma \cdot \begin{cases} \{\varepsilon\} & \text{if } \forall\alpha\ \forall s\ [(\alpha, s) \in X \Rightarrow s = E], \\ \{\partial\} & \text{if } \neg\langle X, \sigma\rangle - U, \tau \to \text{ and } \exists\alpha\ \exists s\ [s \neq E \wedge (\alpha, s) \in X], \\ \bigcup \{\sigma' \cdot F(X')(\sigma') : \langle X, \sigma\rangle - U, \tau \to \langle X', \sigma'\rangle\} & \text{otherwise,} \end{cases}$$

where

$$\langle X, \sigma\rangle - U, \tau \to = \exists X'\ \exists\sigma'\ [\langle X, \sigma\rangle - U, \tau \to \langle X', \sigma'\rangle].$$

Now the operational semantics $\mathcal{O}_U : \mathcal{P}_{fin}(\mathit{LStat}) \to P$ is given as

$$\mathcal{O}_U = \text{Fixed point } (\Phi_U).$$

It is straightforward to prove that Φ_U is a contraction and thus has a unique fixed point.

The definition of Φ_U is very similar to the definition of Φ in the previous section (Definition 2.5). If, for a given $X \in \mathcal{P}_{fin}(LStat)$ and $\sigma \in \Sigma$, we have that $\neg\langle X, \sigma\rangle - U, \tau \rightarrow$, then no computation steps, which are indicated by τ, are possible from $\langle X, \sigma\rangle$. The transitions that *are* possible are of the form

$$\langle X, \sigma\rangle - U, (\alpha, \beta_1 ! m(\beta_2)) \rightarrow \rho \quad \text{or} \quad \langle X, \sigma\rangle - U, (\alpha ? m) \rightarrow \rho',$$

denoting attempts of a single object α to perform a communication action without any matching object being present. This is an instance of deadlock and therefore we here have that $\mathcal{O}_U[\![X]\!](\sigma) = \{\partial\}$. On the other hand, for every transition

$$\langle X, \sigma\rangle - U, \tau \rightarrow \langle X', \sigma'\rangle$$

the set $\mathcal{O}_U[\![X]\!](\sigma)$ includes the set $\sigma' \cdot \mathcal{O}_U[\![X']\!](\sigma')$, in which the transformed state σ' is concatenated with the operational meaning of X' in state σ'.

Finally, we can give the operational semantics of a unit.

DEFINITION 4.11 (Operational semantics of a unit). Let $[\![\cdots]\!]_{\mathcal{O}} : Unit \rightarrow P$ be given, for a unit $U = \langle(\cdots, C_n \Leftarrow s_n), \cdots\rangle$, by

$$[\![U]\!]_{\mathcal{O}} = \mathcal{O}_U[\![\{(\nu(\varnothing), s_n)\}]\!].$$

The execution of a unit $U = \langle(\cdots, C_n \Leftarrow s_n), \cdots\rangle$ consists of the creation of an object of class C_n and the execution of its body. Its name is given by $\nu(\varnothing)$, the name of the first object.

Comparison with [ABKR86(a)]. In [ABKR86(a)], an operational semantics for POOL is defined which differs from $\mathcal{O}_U$ in a number of respects. There, a transition relation without labels is used whereas we have a labelled transition relation here; further, in [ABKR86(a)] communication is modelled by means of a so-called *wait* statement as opposed to the release statement we use here; also our use of parameterized expressions and statements is new. All these differences can be seen as minor variations of the semantic definitions and are motivated by the main goal of this paper, which is to relate the operational semantics with the denotational one. There is one major difference, however, which we shall treat in some detail: In Definition 4.10 of this paper, $\mathcal{O}_U$ is given as the fixed point of a contraction, whereas in [ABKR86(a)] the operational semantics is defined in terms of finite and infinite sequences of transitions. In order to show the equivalence of both approaches, we now define an operational semantics $\mathcal{O}^*_U$ in the style of [ABKR86(a)], for which we next shall prove that it equals $\mathcal{O}_U$.

DEFINITION 4.12 (Alternative operational semantics). Let, for a $U \in Unit$, the function

$$\mathcal{O}^*_U : \mathcal{P}_{fin}(LStat) \rightarrow P$$

be given as follows. Let $X \in \mathcal{P}_{fin}(LStat)$ and $\sigma \in \Sigma$. We put for a word $w \in \Sigma^\infty_\partial$:

$$w \in \mathcal{O}^*_U[\![X]\!](\sigma)$$

if and only if one of the following conditions is satisfied:

(1) $w = \sigma_1 \cdots \sigma_n$ and there exist $X_1, \cdots, X_n$ such that

$$\langle X, \sigma\rangle - U, \tau \rightarrow \langle X_1, \sigma_1\rangle - U, \tau \rightarrow \cdots - U, \tau \rightarrow \langle X_n, \sigma_n\rangle \quad \text{and} \quad \forall(\alpha, s) \in X_n[s = E].$$

(2) $w = \sigma_1\sigma_2 \cdots$ and there exist $X_1, X_2, \cdots$ such that

$$\langle X, \sigma\rangle - U, \tau \rightarrow \langle X_1, \sigma_1\rangle - U, \tau \rightarrow \langle X_2, \sigma_2\rangle - U, \tau \rightarrow \cdots.$$

(3) $w = \sigma_1 \cdot \cdots \cdot \sigma_n \cdot \partial$ and there exist $X_1, \cdots, X_n$ such that

$$\langle X, \sigma\rangle - U, \tau \to \langle X_1, \sigma_1\rangle - U, \tau \to \cdots - U, \tau \to \langle X_n, \sigma_n\rangle \quad \text{and}$$

$$\exists(\alpha, s) \in X_n[s \neq E] \quad \text{and} \quad \neg\langle X_n, \sigma_n\rangle - U, \tau \to .$$

It is not straightforward that the sets $\mathcal{O}_U^*[\![X]\!](\sigma)$ are in P, that is, that they are compact; we prove this fact in the following lemma.

LEMMA 4.13 (Compactness of $\mathcal{O}_U^*$). *For every $X \in \mathcal{P}_{fin}(LStat)$ and $\sigma \in \Sigma$: $\mathcal{O}_U^*[\![X]\!](\sigma)$ is compact.*

Proof. Let $(w_i)_i$ be a sequence of words in $\mathcal{O}_U^*[\![X]\!](\sigma)$ $(\subseteq \Sigma_\partial^\infty)$, say

$$w_i = \sigma_i^1 \sigma_i^2 \sigma_i^3 \cdots .$$

We show that $(w_i)_i$ has a converging subsequence with its limit in $\mathcal{O}_U^*[\![X]\!](\sigma)$. Assume for simplicity that all words w_i are infinite. Since $w_i \in \mathcal{O}_U^*[\![X]\!](\sigma)$, for every i, there exist infinite transition sequences such that

$$\langle X, \sigma\rangle \to \langle X_i^1, \sigma_i^1\rangle \to \langle X_i^2, \sigma_i^2\rangle \to \cdots$$

(omitting the labels U, τ). From the definition of $\to$ it follows that the set

$$\{\langle X', \sigma'\rangle : \langle X, \sigma\rangle \to \langle X', \sigma'\rangle\}$$

is finite. (This follows from the observation that according to the axioms only a finite number of transitions is possible from an arbitrary configuration; this property is preserved by all the rules.) Thus there exists a pair $\langle X_1, \sigma_1\rangle$ such that for infinitely many i's:

$$\langle X_i^1, \sigma_i^1\rangle = \langle X_1, \sigma_1\rangle.$$

Let $f_1 : \mathbb{N} \to \mathbb{N}$ be a monotonic function with, for all i,

$$\langle X_{f(i)}^1, \sigma_{f(i)}^1\rangle = \langle X_1, \sigma_1\rangle.$$

Next we proceed with the subsequence $(w_{f_1(i)})_i$ of $(w_i)_i$ and repeat the above argument, now with respect to the set

$$\{\langle X', \sigma'\rangle : \langle X_1, \sigma_1\rangle \to \langle X', \sigma'\rangle\}.$$

Continuing in this way, we find a sequence of monotonic functions $(f_k)_k$, defining a sequence of subsequences of $(w_i)_i$, and a sequence of configurations $(\langle X_k, \sigma_k\rangle)_k$ such that

$$\forall k\, \forall j\, \forall i \leqq k[\sigma_{f_k(j)}^i = \sigma_i] \quad \text{and} \quad \langle X, \sigma\rangle \to \langle X_1, \sigma_1\rangle \to \langle X_2, \sigma_2\rangle \to \cdots$$

and, moreover, such that the sequence $(w_{f_{k+1}(i)})_i$ is a subsequence of the sequence of $(w_{f_k(i)})_i$. Now we define

$$g(i) = f_i(i).$$

Then we have

$$\lim_{i \to \infty} w_{g(i)} = \sigma_1 \sigma_2 \sigma_3 \cdots .$$

Thus we have constructed a converging subsequence of $(w_i)_i$ with its limit in $\mathcal{O}_U^*[\![U]\!](\sigma)$. (In case the words w_i are not all infinite a similar argument can be given.)

It is not difficult to show that $\mathcal{O}_U = \mathcal{O}_U^*$.

THEOREM 4.14. $\mathcal{O}_U = \mathcal{O}_U^*$.

Proof. We prove that $\mathcal{O}_U^*$ is also a fixed point of Φ_U, from which the equality follows. Let $X \in \mathcal{P}_{fin}(LStat)$ such that there exist $(\alpha, s) \in X$ $[s \neq E]$, let $\sigma \in \Sigma$ and let $w \in \Sigma_\partial^\infty$. If $w = \partial$ then

$$w \in \Phi_U(\mathcal{O}_U^*)(X)(\sigma) \Leftrightarrow w \in \mathcal{O}_U^*[\![X]\!](\sigma).$$

Now suppose $w \neq \partial$. We have

$$\begin{aligned} w \in \mathcal{O}_U^*[\![X]\!](\sigma) &\Leftrightarrow \exists \sigma' \in \Sigma \quad \exists X' \in \mathcal{P}_{fin}(LStat) \quad \exists w' \in \Sigma_\partial^\infty \\ &\qquad [\langle X, \sigma\rangle \to \langle X', \sigma'\rangle \wedge w = \sigma' \cdot w' \wedge w' \in \mathcal{O}_U^*[\![X']\!](\sigma')] \\ &\Leftrightarrow [\text{definition } \Phi_U] \\ &\qquad w \in \Phi_U(\mathcal{O}_U^*)(X)(\sigma). \end{aligned}$$

So we see that $\mathcal{O}_U^* = \Phi_U(\mathcal{O}_U^*)$.

5. A denotational semantics for POOL. The denotational semantics that is defined in this section was already presented (in a slightly different form) in [ABKR86(b)]. (For a comparison of the two models we refer the reader to the end of this section.)

Our denotational model has a so-called *domain* (a solution of a reflexive domain equation) for its semantic universe. In [BZ82] it was first described how to solve these equations in a metric setting. Then, in [AR88], this approach was generalized in order to deal with equations of the following form: $P \cong \cdots P \to \cdots$, a case that was not covered by [BZ82]. For a quick overview of the main results of [AR88], the reader might want to read § 2 of [ABKR86(b)].

Further, our model is based on the use of *continuations*. For an extensive treatment of continuations and expression continuations, which we shall use as well, we refer to [Go79].

We start with the definition of a domain $\bar{P}$, the elements of which we shall call *processes* from now on.

DEFINITION 5.1 (Semantic process domain $\bar{P}$). Let $(p, q \in)P$ be a complete ultra metric space satisfying the following reflexive domain equation:

$$P \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{compact}(Step_P)),$$

where $(\pi, \rho \in)Step_P$ is

$$Step_P = Comp_P \cup Send_P \cup Answer_P,$$

with

$$\begin{aligned} Comp_P &= \Sigma \times P, \\ Send_P &= Obj \times MName \times Obj \times (Obj \to P) \times P, \\ Answer_P &= Obj \times MName \times (Obj \to (Obj \to P) \to {}^1P). \end{aligned}$$

(The sets $\{p_0\}$, Σ, *Obj*, and *MName* become complete ultra-metric spaces by supplying them with the discrete metric.)

In [AR88], it is described how to find for such an equation a solution that is unique up to isomorphy. Let us try to explain intuitively the intended interpretation of the domain $\bar{P}$. First, we observe that in the equation above the subexpression $id_{1/2}$ (defined in Appendix I, I.6(e)) is necessary only to guarantee that the equation is solvable by defining a contracting functor on $\mathcal{C}$, the category of complete metric spaces. For a more operational understanding of the equation, for example, it does not matter.

A process $p \in \bar{P}$ is either p_0 or a function from Σ to $\mathcal{P}_{compact}(Step_{\bar{P}})$, the set of all *compact* subsets of $Step_{\bar{P}}$. The process p_0 is the terminated process. For $p \neq p_0$, the process p has the choice, depending on the current state σ, among the *steps* in the set $p(\sigma)$. If $p(\sigma) = \emptyset$, then no further action is possible, which is interpreted as abnormal termination. For $p(\sigma) \neq \emptyset$, each step $\pi \in p(\sigma)$ consists of some action (for instance, a change of the state σ or an attempt at communication) and a *resumption* of this action; that is to say, the remaining actions to be taken after this action. There are three different types of steps $\pi \in Step_{\bar{P}}$.

First, a step may be an element of $\Sigma \times \bar{P}$, say

$$\pi = \langle \sigma', p' \rangle.$$

The only action is a change of state: σ' is the new state. Here the process p' is the resumption, indicating the remaining actions process p can do. (When $p' = p_0$ no steps can be taken after this step π.)

Second, π might be a *send step*, $\pi \in Send_{\bar{P}}$. In this case we have, say

$$\pi = \langle \alpha, m, \beta, f, p \rangle,$$

with $\alpha \in Obj$, $m \in MName$, $\beta \in Obj$, $f \in (Obj \to \bar{P})$, and $p \in \bar{P}$. The action involved here consists of an attempt at communication, in which a message is sent to the object α, specifying the method m, together with the parameter β. This is the interpretation of the first three components α, m, and β. The fourth component f, called the *dependent* resumption of this send step, indicates the steps that will be taken after the sender has received the result of the message. These actions will depend on the result, which is modelled by f being a function that yields a process when it is applied to an object name (the result of the message). The last component p, called the *independent* resumption of this send step, represents the steps to be taken after this send step that need *not* wait for the result of the method execution.

Finally, π might be an element of $Answer_{\bar{P}}$, say

$$\pi = \langle \alpha, m, g \rangle$$

with $\alpha \in Obj$, $m \in MName$, and $g \in (Obj \to (Obj \to \bar{P}) \to {}^1\bar{P})$. It is then called an *answer step*. The first two components of π express that the object α is willing to accept a message that specifies the method m. The last component g, the resumption of this answer step, specifies what should happen when an appropriate message actually arrives. The function g is then applied to the parameter in this message and to the dependent resumption of the sender (specified in its corresponding send step). It then delivers a process which is the resumption of the sender and the receiver *together*, which is to be composed in parallel with the independent resumption of the send step.

We now define a semantic operator for the *parallel composition* (or *merge*) of two processes, for which we shall use the symbol $\|$. It is *auxiliary* in the sense that it does not correspond to a syntactic operator in the language POOL.

DEFINITION 5.2 (Parallel composition). Let $\| : \bar{P} \times \bar{P} \to \bar{P}$ be such that it satisfies the following equation:

$$p \| q = \lambda\sigma \cdot ((p(\sigma) \mathbin{\lfloor\!\lfloor} q) \cup (q(\sigma) \mathbin{\lfloor\!\lfloor} p) \cup (p(\sigma) \mid_\sigma q(\sigma))),$$

for all $p, q \in \bar{P} \setminus \{p_0\}$, and such that $p_0 \| q = q \| p_0 = p_0$. Here, $X \mathbin{\lfloor\!\lfloor} q$ and $X \mid_\sigma Y$ are defined by

$$X \mathbin{\lfloor\!\lfloor} q = \{\pi \mathbin{\hat{\|}} q : \pi \in X\},$$

$$X \mid_\sigma Y = \bigcup \{\pi \mid_\sigma \rho : \pi \in X, \rho \in Y\},$$

where $\pi \hat{\|} q$ is given by

$$\langle \sigma', p' \rangle \hat{\|} q = \langle \sigma', p' \| q \rangle,$$
$$\langle \alpha, m, \beta, f, p \rangle \hat{\|} q = \langle \alpha, m, \beta, f, p \| q \rangle, \quad \text{and}$$
$$\langle \alpha, m, g \rangle \hat{\|} q = \langle \alpha, m, \lambda\beta \cdot \lambda h \cdot (g(\beta)(h) \| q) \rangle,$$

and $\pi |_\sigma \rho$ by

$$\pi |_\sigma \rho = \begin{cases} \{\langle \sigma, g(\beta)(f) \| p \rangle\} & \text{if } \pi = \langle \alpha, m, \beta, f, p \rangle \text{ and } \rho = \langle \alpha, m, g \rangle \\ & \text{or } \rho = \langle \alpha, m, \beta, f, p \rangle \text{ and } \pi = \langle \alpha, m, g \rangle, \\ \varnothing & \text{otherwise.} \end{cases}$$

We observe that this definition is self-referential, since the merge operator occurs at the right-hand side of the definition. For a formal justification of this definition see the appendix of [ABKR86(b)], where the merge operator is given as the unique fixed point of a contraction on $\bar{P} \times \bar{P} \to {}^1\bar{P}$.

Since we intend to model parallel composition by interleaving, the merge of two processes p and q consists of three parts. The first part contains all possible first steps of p followed by the parallel composition of their respective resumptions with q. The second part contains similarly the first steps of q. The last part contains the communication steps that result from two matching communication steps taken simultaneously by process p and q. For $\pi \in Step_{\bar{P}}$ the definition of $\pi \hat{\|} q$ is straightforward. The definition of $\pi |_\sigma \rho$ is more involved. It is the empty set if π and ρ do not match. Now suppose they do match, say $\pi = \langle \alpha, m, \beta, f, p \rangle$ and $\rho = \langle \alpha, m, g \rangle$. Then π is a *send* step, denoting a request to object α to execute the method m, and ρ is an *answer* step, denoting that the object α is willing to accept a message that requests the execution of the method m. In $\pi |_\sigma \rho$, the state σ remains unaltered. Since g, the third component of ρ, represents the meaning of the execution of the method m, it needs the parameter β that is specified by α. Moreover, g depends on the dependent resumption f of the send step π. This explains why both β and f are supplied as arguments to the function g. Now it can be seen that $g(\beta)(f) \| p$ represents the resumption of the sender and the receiver together. (In order to get more insight into this definition it is advisable to return to it after having seen the definition of the semantics of an answer statement.)

The merge operator is associative, which can easily be proved as follows. Define

$$\varepsilon = \sup_{p,q,r \in \bar{P}} \{d_{\bar{P}}((p \| q) \| r, p \| (q \| r))\}.$$

Then, using the fact that the operator $\|$ satisfies the equation above, one can show that $\varepsilon \leqq \frac{1}{2} \cdot \varepsilon$. Therefore $\varepsilon = 0$, and $\|$ is associative.

Now we come to the definition of the semantics of expressions and statements. We specify a pair of functions $\langle \mathscr{D}_E, \mathscr{D}_S \rangle$ of the following type:

$$\mathscr{D}_E : L_E \to AObj \to Cont_E \to {}^1\bar{P}, \qquad \mathscr{D}_S : L_S \to AObj \to Cont_S \to {}^1\bar{P}$$

where

$$Cont_E = Obj \to \bar{P} \quad \text{and} \quad Cont_S = \bar{P}.$$

Let $s \in L_S$, $\alpha \in AObj$, and $p \in \bar{P}$. The semantic value of the statement s is given by

$$\mathscr{D}_S[\![s]\!](\alpha)(p).$$

The object name α represents the object that executes s. Second, the semantic value of s depends on its so-called *continuation* p: the semantic value of everything that will happen after the execution of s. The main advantage of the use of continuations is that it enables us to describe the semantics of expressions in a concise and elegant way.

The semantic value of an expression $e \in L_E$, for an object α and an expression continuation $f \in Cont_E$, is given by

$$\mathscr{D}_E[\![e]\!](\alpha)(f).$$

The evaluation of an expression e always results in a value (an element of *Obj*), on which the continuation of such an expression generally depends. The function f, when applied to the result β of e, will yield a process $f(\beta) \in \bar{P}$ that is to be executed after the evaluation of e.

Please note the difference between the notions of *resumption* and *continuation.* A resumption is a part of a semantic step $\pi \in Step_{\bar{P}}$, indicating the remaining steps to be taken after the current one. A continuation, on the other hand, is an argument to a semantic function. It may appear as a resumption in the result. A good example of this is the definition of $\hat{F}_S(x \leftarrow e)$ (in Definition 5.3(SI)) below.

DEFINITION 5.3 (Semantics of expressions and statements). Let

$$Q_E = L_E \to AObj \to Cont_E \to {}^1\bar{P}, \qquad Q_S = L_S \to AObj \to Cont_S \to {}^1\bar{P}.$$

For every unit $U \in Unit$ we define a pair of functions $\mathscr{D}_U = \langle \mathscr{D}_E, \mathscr{D}_S \rangle$ by

$$\mathscr{D}_U = \text{Fixed point } (\Psi_U),$$

where

$$\Psi_U : (Q_E \times Q_S) \to (Q_E \times Q_S)$$

is defined by induction on the structure of L_E and L_S by the following clauses. For $F = \langle F_E, F_S \rangle$ we denote $\Psi_U(F)$ by $\hat{F} = \langle \hat{F}_E, \hat{F}_S \rangle$. Let $p \in Cont_S = \bar{P}$, $f \in Cont_E = Obj \to \bar{P}$ and $\alpha \in AObj$. Then:

EXPRESSIONS:

(E1, instance variable) $\qquad \hat{F}_E(x)(\alpha)(f) = \lambda\sigma \cdot \{\langle \sigma, f(\sigma_1(\alpha)(x)) \rangle\}.$

The value of the instance variable x is looked up in the first component of the state σ supplied with the name α of the object that is evaluating the expression. The continuation f is then applied to the resulting value.

(E2, temporary variable) $\qquad \hat{F}_E(u)(\alpha)(f) = \lambda\sigma \cdot \{\langle \sigma, f(\sigma_2(\alpha)(u)) \rangle\}.$

(E3, send expression)

$$\hat{F}_E(e_1 ! m(e_2))(\alpha)(f) = \hat{F}_E(e_1)(\alpha)(\lambda\beta_1 \cdot \hat{F}_E(e_2)(\alpha)(\lambda\beta_2 \cdot \lambda\sigma \cdot \{\langle \beta_1, m, \beta_2, f, p_0 \rangle\})).$$

The expressions e_1 and e_2 are evaluated successively. Their results correspond to the formal parameters β_1 and β_2 of their respective continuations. Finally, a send step is performed. The object name β_1 refers to the object to which the message is sent; β_2 represents the parameter for the execution of the method m. Besides these values and the method name m, the final step $\langle \beta_1, m, \beta_2, f, p_0 \rangle$ also contains the expression continuation f of the send expression as the dependent resumption. If the attempt at communication succeeds, this continuation will be supplied with the result of the method execution. The independent resumption of this send step is initialized at p_0.

(E4, new-expression) $\qquad \hat{F}_E(\textbf{new}\,(C))(\alpha)(f) = \lambda\sigma \cdot \{\langle \sigma', f(\beta) \parallel F_S(s_C)(\beta)(p_0) \rangle\},$

where

$$\beta = \nu(\sigma_3),$$

$$\sigma' = \langle \sigma_1, \sigma_2, \sigma_3 \cup \{\beta\} \rangle, \qquad C \Leftarrow s_C \in U.$$

A new object of class C is created. It is called $\nu(\sigma_3)$: the function ν supplied with the set of all object names currently in use yields a name that is not yet being used. The state σ is changed by expanding the set σ_3 with the new name β. The process $F_S(s_C)(\beta)(p_0)$ is the meaning of the body of the new object β with p_0 as a nil continuation. It is composed in parallel with $f(\beta)$, the process resulting from the application of the continuation f to β, the result of the evaluation of this new-expression. We are able to perform this parallel composition because we know from f what should happen after the evaluation of this new-expression, so here the use of continuations is essential.

(E5, sequential composition) $\quad \hat{F}_E(s\,;\,e)(\alpha)(f) = \hat{F}_S(s)(\alpha)(\hat{F}_E(e)(\alpha)(f))$.

The continuation of s is the execution of e followed by f. Note that a semantic operator for sequential composition is absent: the use of continuations has made it superfluous.

(E6, **self**) $\quad \hat{F}_E(\textbf{self})(\alpha)(f) = f(\alpha)$.

The continuation of f is supplied with the value of the expression **self**, that is, the name of the object executing this expression. We use $f(\alpha)$ instead of $\lambda\beta \cdot \{\langle\sigma, f(\alpha)\rangle\}$ in this definition wishing to express that the value of self is immediately present: it does not take a step to evaluate it.

STATEMENTS:

(S1, assignment to an instance variable)

$$\hat{F}_S(s \leftarrow e)(\alpha)(p) = \hat{F}_E(e)(\alpha)(\lambda\beta \cdot \lambda\sigma \cdot \{\langle\sigma', p\rangle\}),$$

where $\sigma' = \sigma\{\beta/\alpha, x\}$. The expression e is evaluated and the result β is assigned to x.

(S2, assignment to a temporary variable)

$$\hat{F}_S(u \leftarrow e)(\alpha)(p) = \hat{F}_E(e)(\alpha)(\lambda\beta \cdot \lambda\sigma \cdot \{\langle\sigma', p\rangle\}),$$

where $\sigma' = \sigma\{\beta/\alpha, u\}$.

(S3, answer statement) $\quad \hat{F}_S(\textbf{answer}\ m)(\alpha)(p) = \lambda\sigma \cdot \{\langle\alpha, m, g_m\rangle\}$,

where

$$g_m = \lambda\beta \cdot \lambda f \cdot \lambda\hat{\sigma} \cdot \{\langle\sigma', F_E(e_m)(\alpha)(\lambda\beta' \cdot \lambda\bar{\sigma} \cdot \{\langle\bar{\sigma}', f(\beta') \parallel p\rangle\})\},$$

with

$$\sigma' = \hat{\sigma}\{\beta/\alpha, u_m\},$$
$$\bar{\sigma}' = \bar{\sigma}\{\hat{\sigma}_2(\alpha)(u_m)/\alpha, u_m\},$$
$$m \Leftarrow \langle u_m, e_m\rangle \in U.$$

The function g_m represents the execution of the method m followed by its continuation. This function g_m expects a parameter β and an expression continuation f, both to be received from an object sending a message specifying the method m. The execution of the method m consists of the evaluation of the expression e_m, which is used in the definition of m, preceded by a state transformation in which the temporary variable u_m is initialized at the value β. After the execution of e, this temporary variable is set back to its old value again. Next, both the continuation of the sending object, supplied with the result β' of the execution of the method m, and the given continuation p are to be executed in parallel. This explains the last resumption: $f(\beta') \parallel p$.

Now that we have defined the semantics of send expressions and answer statements let us briefly return to the definition of $\pi|_\sigma \rho$ (Definition 5.2). Let $\pi = \langle \alpha, m, \beta, f, q\rangle$ (the result from the elaboration of a send expression) and $\rho = \langle \alpha, m, g\rangle$ (resulting from an answer statement). Then $\pi|_\sigma \rho$ is defined as

$$\pi|_\sigma \rho = \{\langle \sigma, g(\beta)(f) \parallel q\rangle\}.$$

We see that the execution of the method m proceeds in parallel with the independent resumption q of the sender. Now we know how g is defined we have

$$g(\beta)(f) = \lambda\sigma \cdot \{\langle \sigma', F_E(e_m)(\alpha)(\lambda\beta' \cdot \lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', f(\beta') \parallel p\rangle\})\rangle\}.$$

The continuation of the execution of m is given by $\lambda\beta' \cdot \lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', f(\beta') \parallel p\rangle\}$, which consists of a state transformation followed by the parallel composition of the continuations f and p. This represents the fact that after the rendez-vous, during which the method is executed, the sender and the receiver of the message can proceed in parallel again. (Of course, the independent resumption q may still be executing at this point.) Moreover, the result β' of the method execution is passed on to the continuation f of the send expression.

(S4, sequential composition) $\quad \hat{F}_S(s_1\,;\, s_2)(\alpha)(p) = \hat{F}_S(s_1)(\alpha)(\hat{F}_S(s_2)(\alpha)(p))$.

(S5, conditional)

$$\hat{F}_S(\textbf{if } e \textbf{ then } s_1 \textbf{ else } s_2 \textbf{ fi})(\alpha)(p) = \hat{F}_E(e)(\alpha)(\lambda\beta \cdot \text{if } \beta = tt \text{ then } \hat{F}_S(s_1)(\alpha)(p) \text{ else } \hat{F}_S(s_2)(\alpha)(p) \text{ fi}).$$

(S6, loop statement)

$$\hat{F}_S(\textbf{do } e \textbf{ then } s \textbf{ od})(\alpha)(p) = \lambda\sigma \cdot \{\langle \sigma, \hat{F}_E(e)(\alpha)(\lambda\beta) \cdot \text{if } \beta = tt \text{ then } \hat{F}_S(s)(\alpha)(F_S(\textbf{do } e \textbf{ then } s \textbf{ od})(\alpha)(p)) \text{ else } p \text{ fi})\rangle\}.$$

(End of Definition 5.3.)

It is not difficult to prove that Ψ_U is a contraction and hence has a unique fixed point $\mathscr{D}_U$. As a matter of fact, we have defined Ψ_U such that it satisfies this property. Note that the original functions F_E and F_S have been used in only three places: in the definition of the semantics of a new-expression, of an answer statement, and of a loop statement. Here the syntactic complexity of the defining part is not necessarily less than that of what is being defined. At those places, we have ensured that the definition is "guarded" by some step $\lambda\sigma \cdot \{\langle \sigma', \cdots\rangle\}$. It is easily verified that in this manner the contractiveness of Ψ_U is indeed implied.

DEFINITION 5.4 (Denotational semantics of a unit). We define $[\![\cdots]\!]_{\mathscr{D}} : \mathit{Unit} \to \bar{P}$. For a unit $U \in \mathit{Unit}$, with $U = \langle(\cdots, C_n \Leftarrow s_n), \cdots\rangle$, we set

$$[\![U]\!]_{\mathscr{D}} = \mathscr{D}_S[\![s_n]\!](\nu(\varnothing))(p_0).$$

The execution of a unit always starts with the creation of an object of class C_n and the execution of its body. Therefore, the meaning of a unit U is given by the

denotational meaning of s_C, the body of class C_n, supplied with $\nu(\varnothing)$, denoting the name of the first active object, and with p_0, the empty continuation.

Comparison with [ABKR86(b)]. There are some differences between the denotational semantics $\langle \mathscr{D}_E, \mathscr{D}_S \rangle$ presented here and the denotational semantics given in [ABKR86(b)]: The former model is given as the fixed point of a contraction Ψ_U and does not use so-called *environments* to deal with process creation (**new** (C)) and the meaning of the execution of a method (**answer** m); the latter model is defined without the use of a contraction and *does* use environments. In [ABKR86(b)], the semantics of a unit U is given with the help of a special environment γ_U, which contains information about the class and method definitions in U and is obtained as the fixed point of a suitably defined contraction. Another difference is the way the loop statement is treated: In this paper, the definition of its semantics fits smoothly in the definition of $\langle \mathscr{D}_E, \mathscr{D}_S \rangle$ as a fixed point. In [ABKR86(b)], a contraction is defined especially for this case.

Another way to express these differences is that the three constructs for recursion present in POOL (i.e., the new expression, the answer, and the loop statement) are treated here by means of one fixed-point definition, whereas in [ABKR86(b)], environments are used for the first two forms of recursion and a specially defined contraction for the last one. However, we state (without proof) that the two definitions are equivalent: it is straightforward how to translate the one approach into the other.

An additional difference between the denotational semantics of a unit given here and the one presented in [ABKR86(b)] is the presence of a semantic representation of the standard objects in the latter, whereas these are not treated in this section. As mentioned before, we do not treat standard objects now because we want to concentrate on the correctness proof. In order to show, however, that our proof (to be presented in § 7) can also deal with standard objects, we shall extend, in Appendix III, both our operational and our denotational semantics with a semantic representation of standard objects, and prove that the correctness result still holds for these extended models.

6. An intermediate semantics. After having defined an intermediate semantics $\mathscr{O}_U$ for $\mathscr{P}_{fin}(LStat)$ and a denotational semantics $\mathscr{D}_U$ for L_E and L_S we shall, in the next section, discuss the relationship between the two. As we did in § 2, we shall compare $\mathscr{O}_U$ and $\mathscr{D}_U$ by relating both to an intermediate semantics $\mathscr{O}'_U : \mathscr{P}_{fin}(LStat) \to \bar{P}$, the definition of which is the subject of this section.

DEFINITION 6.1 (Intermediate semantics $\mathscr{O}'_U$). Let $U \in Unit$. Let $\mathscr{O}'_U : \mathscr{P}_{fin}(LStat) \to \bar{P}$ be given by

$$\mathscr{O}'_U = \text{Fixed Point}\,(\Phi'_U),$$

where

$$\Phi'_U : (\mathscr{P}_{fin}(LStat) \to \bar{P}) \to (\mathscr{P}_{fin}(LStat) \to \bar{P})$$

is defined, for $F \in \mathscr{P}_{fin}(LStat) \to \bar{P}$ and $X \in \mathscr{P}_{fin}(LStat)$, as follows.

If for all α and s $[(\alpha, s) \in X \Rightarrow s = E]$, then $\Phi'_U(F)(X) = p_0$. Otherwise we have

$$\Phi'_U(F)(X) = \lambda\sigma \cdot (C_F \cup S_F \cup A_F)$$

where

$$C_F = \{\langle \sigma', F(X') \rangle : \langle X, \sigma \rangle - U, \tau \to \langle X', \sigma' \rangle\},$$

$$S_F = \{\langle \beta_1, m, \beta_2, \lambda\beta \cdot F(\{(\alpha, \psi(\beta))\}), F(X') \rangle : \langle X, \sigma \rangle - U, (\alpha, \beta_1\, !m(\beta_2)) \to \langle \{(\alpha, \psi)\} \cup X', \sigma \rangle\},$$

$$A_F = \{\langle \alpha, m, g_m \rangle : \langle X, \sigma \rangle - U, (\alpha\, ?\, m) \to \langle \{(\alpha, s)\} \cup X', \sigma \rangle\}$$

with

$$g_m = \lambda\beta \cdot \lambda f \cdot (\lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta' \cdot \lambda\hat{\sigma} \cdot \{\langle \hat{\sigma}', f(\beta') \parallel F(\{\{(\alpha, s)\}\})\rangle\})\rangle\} \parallel F(X')),$$

and

$$\bar{\sigma}' = \bar{\sigma}\{\beta/\alpha, u_m\}, \qquad \hat{\sigma}' = \hat{\sigma}\{\bar{\sigma}_2(\alpha)(u_m)/\alpha, u_m\},$$
$$m \Leftarrow \langle u_m, e_m \rangle \in U.$$

(It is straightforward to show that Φ'_U is a contraction.)

The function $\mathcal{O}'_U$ differs from the operational semantics $\mathcal{O}_U$ in two ways. First, its range is the semantic universe $\bar{P}$, which is used for the denotational semantics $\mathcal{D}_U$, instead of P, the semantic universe of $\mathcal{O}_U$: For every set $X \in \mathcal{P}_{fin}(LStat)$ the function $\mathcal{O}'_U$ yields a *process* $\mathcal{O}'_U(X) \in \bar{P}$, rather than a function from states to sets of streams of states. Second, in addition to the computation steps (indicated by the set C_F above) single-sided communication steps are present in $\mathcal{O}'_U(X)$ (indicated by S_F and A_F, for send and answer steps), whereas $\mathcal{O}_U(X)$ contains only computation steps. On the other hand, the similarity between the definitions of $\mathcal{O}_U$ and $\mathcal{O}'_U$ is obvious: both are based on the transition relation $-U\rightarrow$ for $\mathcal{O}_{fin}(LStat)$.

At first sight, two facts regarding the relation between $\mathcal{O}'_U$ and $\mathcal{D}_U$ can be mentioned. First, they have the same range, that is, the semantic universe $\bar{P}$ of processes, in which single-sided communication actions are visible. Second, $\mathcal{D}_U$ is defined compositionally with the use of semantic operators (like the merge $\parallel$), whereas the definition of $\mathcal{O}'_U$ is based, as was mentioned above, on the transition relation $-U\rightarrow$.

In the next section the relationship between $\mathcal{O}_U$, $\mathcal{O}'_U$, and $\mathcal{D}_U$ will be formally expressed. Let us, for the time being, try to elucidate the definition of $\mathcal{O}'_U$ above by explaining what communication steps are present in $\mathcal{O}'_U(X)$.

Corresponding with every send transition of the form

$$\langle X, \sigma\rangle - U, (\alpha, \beta_1 ! m(\beta_2)) \rightarrow \langle \{(\alpha, \psi)\} \cup X', \sigma\rangle$$

the set $\mathcal{O}'_U(X)(\sigma)$, for a state $\sigma \in \Sigma$, contains a send step of the form

$$\langle \beta_1, m, \beta_2, \lambda\beta \cdot \mathcal{O}'_U(\{(\alpha, \psi(\beta))\}), \mathcal{O}'_U(X')\rangle.$$

Here β_1, m, and β_2 indicate that a message specifying the method m with parameter β_2 is sent to the object β_1. The dependent resumption of this send step is $\lambda\beta \cdot \mathcal{O}'_U(\{(\alpha, \psi(\beta))\})$: the meaning of the statement that will be executed by α as soon as it receives the result β of the message. The last component of this send step, the independent resumption, consists of $\mathcal{O}'_U(X')$, which is the meaning of all the statements executed by objects other than α. Thus it is reflected that these objects need not wait till the message is answered; they may proceed in parallel.

Next, $\mathcal{O}'_U(X)(\sigma)$ can contain some answer steps. For every answer transition

$$\langle X, \sigma\rangle - U, (\alpha\, ?\, m) \rightarrow \langle \{(\alpha, s)\} \cup X', \sigma\rangle$$

the set $\mathcal{O}'_U(X)(\sigma)$ includes an answer step

$$\langle \alpha, m, g_m \rangle,$$

with g_m as in the definition above. It indicates that the object α is willing to answer a message specifying the method m, while the resumption g_m indicates what should happen when an appropriate message arrives. This function g_m, when supplied with a parameter β and a dependent resumption f (both to be received from the sending object), consists of the parallel composition of the process $\mathcal{O}'_U(X')$ together with the process

$$\lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta' \cdot \lambda\hat{\sigma} \cdot \{\langle \hat{\sigma}', f(\beta') \parallel \mathcal{O}'_U(\{(\alpha, s)\})\rangle\})\rangle\}.$$

(Note that we have used the function $\mathscr{D}_E$ here; the definition of $\mathscr{O}'_U$ therefore depends on its definition.) The process $\mathscr{O}'_U(X')$ stands for the meaning of all the statements executed by objects other than the object α: these objects may proceed in parallel with the execution of the method m, the meaning of which is indicated by the second process. Its interpretation is the same as in the definition of $\mathscr{D}_S[\![\textbf{answer } m]\!](\alpha)(p)$ in the previous section but for the fact that here the last resumption of this process consists of $f(\beta') \parallel \mathscr{O}'_U(\{(\alpha, s)\})$: the parallel composition of the dependent resumption of the sender (supplied with the result β' of the method m) and the meaning of the statement s, with which the object α will continue after it has answered the message.

7. Semantic correctness. We are now ready to establish the main result of this paper. We shall relate the operational semantics $\mathscr{O}_U$ and the denotational semantics $\mathscr{D}_U$ by first comparing $\mathscr{O}_U$ and $\mathscr{O}'_U$, the intermediate semantics defined in the previous section, and next comparing $\mathscr{O}'_U$ and $\mathscr{D}_U$. These relationships will be formally expressed by means of suitably defined abstraction operations. From this we shall deduce the fact that

$$[\![U]\!]_{\mathscr{O}} = \mathit{abstr}([\![U]\!]_{\mathscr{D}}),$$

where $\mathit{abstr}: \bar{P} \to P$ is such an abstraction operation.

7.1. Comparing $\mathscr{O}_U$ and $\mathscr{O}'_U$. We start with the definition of $\mathit{abstr}: \bar{P} \to P$, which relates the semantic universes P and $\bar{P}$ of $\mathscr{O}_U$ and $\mathscr{O}'_U$.

DEFINITION 7.1 (Abstraction operation *abstr*). Let $\mathit{abstr}: \bar{P} \to P$ be defined as follows. We set $\mathit{abstr}(p_0) = \{\varepsilon\}$. If $p \in \bar{P} \setminus \{p_0\}$, then

$$\mathit{abstr}(p) = \lambda\sigma \cdot \begin{cases} \{\partial\} & \text{if } p(\sigma) \cap \mathit{Comp}_{\bar{P}} = \varnothing, \\ \bigcup \{\sigma' \cdot \mathit{abstr}(p')(\sigma') : \langle \sigma', p' \rangle \in p(\sigma)\} & \text{otherwise,} \end{cases}$$

where $\mathit{Comp}_{\bar{P}} = \Sigma \times \bar{P}$. (Formally, we can define this operation correctly by giving it as the fixed point of a suitably defined contraction on $\bar{P} \to P$: See Appendix II for an extensive formal treatment of the function *abstr*.)

The function *abstr* transforms a process $p \in \bar{P}$ into a function $\mathit{abstr}(p) \in P = \Sigma \to \mathscr{P}_{\mathit{ncompact}}(\Sigma^{\infty}_{\partial})$, which yields for every $\sigma \in \Sigma$ a set $\mathit{abstr}(p)(\sigma)$ of streams. (If one regards the process p as a treelike structure, these streams can be considered the branches of p.) If $p(\sigma) \cap \mathit{Comp}_{\bar{P}} = \varnothing$, that is, if $p(\sigma)$ is empty or contains only single-sided communication steps, then we have a case of deadlock because, operationally, single-sided communication is not possible. Therefore we then have that $\mathit{abstr}(p)(\sigma) = \{\partial\}$. If, however, $p(\sigma)$ does contain a computation step $\langle \sigma', p' \rangle$, then we have that: $\sigma' \cdot \mathit{abstr}(p')(\sigma') \subseteq \mathit{abstr}(p)(\sigma)$. The changed state σ' is concatenated with $\mathit{abstr}(p')(\sigma')$, in which σ' is passed through to *abstr* applied to p', the resumption of $\langle \sigma', p' \rangle$. Thus the effect of different state transformations occurring subsequently in p is accumulated.

Next, we use the operation *abstr* to relate Φ_U and Φ'_U.

THEOREM 7.2 (Relating Φ_U and Φ'_U). *For all* $F \in \mathscr{P}_{\mathit{fin}}(\mathit{LStat}) \to \bar{P}[\Phi_U(\mathit{abstr} \circ F) = \mathit{abstr} \circ (\Phi'_U(F))]$.

Proof. Let $F \in \mathscr{P}_{\mathit{fin}}(\mathit{LStat}) \to \bar{P}$, $X \in \mathscr{P}_{\mathit{fin}}(\mathit{LStat})$, and $\sigma \in \Sigma$. Suppose $\neg\forall\alpha\, \forall s\, [(\alpha, s) \in X \Rightarrow s = E]$. If $\neg\langle X, \sigma\rangle - U, \tau \to$, then

$$\begin{aligned} \Phi_U(\mathit{abstr} \circ F)(X)(\sigma) &= \{\partial\} \\ &= \mathit{abstr}(\Phi'_U(F)(X))(\sigma), \end{aligned}$$

since $\Phi'_U(F)(X)(\sigma) \cap \mathit{Comp}_{\bar{P}} = \varnothing$. (Recall that $\mathit{Comp}_{\bar{P}} = \Sigma \times \bar{P}$.) If $\langle X, \sigma\rangle - U, \tau \to$, we

have

$$\begin{aligned}\Phi_U(abstr \circ F)(X)(\sigma) &= \bigcup\{\sigma' \cdot (abstr \circ F)(X')(\sigma') : \langle X, \sigma\rangle - U, \tau \to \langle X', \sigma'\rangle\} \\ &= \bigcup\{\sigma' \cdot (abstr(F)(X'))(\sigma') : \langle X, \sigma\rangle - U, \tau \to \langle X', \sigma'\rangle\} \\ &= [\text{see Definition 6.1}] \\ &\quad abstr(\lambda\sigma \cdot C_F)(\sigma) \\ &= abstr(\lambda\sigma \cdot (C_F \cup S_F \cup A_F))(\sigma) \\ &= abstr(\Phi'_U(F)(X))(\sigma) \\ &= (abstr \circ \Phi'_U(F))(X)(\sigma).\end{aligned}$$

Since Φ_U and Φ'_U are contractions and thus have unique fixed points, the following corollary is straightforward.

COROLLARY 7.3. $\mathcal{O}_U = abstr \circ \mathcal{O}'_U$.

7.2. Comparing $\mathcal{O}'_U$ and $\mathcal{D}_U$. To compare $\mathcal{O}'_U : \mathcal{P}_{fin}(LStat) \to \bar{P}$ and $\mathcal{D}_U \in Q_E \times Q_S$ we define an extension of $\mathcal{D}_U$ $(=\langle \mathcal{D}_E, \mathcal{D}_S\rangle)$ in two steps. First, we define $\mathcal{D}'_U$ $(=\langle \mathcal{D}'_E, \mathcal{D}'_S\rangle) \in Q'_E \times Q'_S$, with

$$Q'_E = L'_E \to AObj \to Cont_E \to {}^1\bar{P}, \qquad Q'_S = L'_S \to AObj \to Cont_S \to {}^1\bar{P},$$

which is as $\mathcal{D}_U$ but with the extended sets of expressions and statements L'_E and L'_S, for its domain. (Recall that L'_S is used in the definition of $LStat = AObj \times L'_S$.) Next, we extend $\mathcal{D}'_U$ to $\mathcal{D}^*_U : \mathcal{P}_{fin}(LStat) \to \bar{P}$, which takes sets of labelled statements for its arguments.

DEFINITION 7.4 ($\mathcal{D}'_U$). Let $\Psi'_U : (Q'_E \times Q'_S) \to (Q'_E \times Q'_S)$ be defined as follows. For $F = \langle F_E, F_S\rangle$, we denote $\Psi'_U(F)$ by $\bar{F} = \langle \bar{F}_E, \bar{F}_S\rangle$. Let $\alpha \in AObj$, $p \in Cont_S = \bar{P}$ and $f \in Cont_E = Obj \to \bar{P}$. Now $\bar{F}$ is defined similarly to $\Psi_U(F)$ (Definition 5.3) but with the following clauses added:

$$\begin{aligned}&\bar{F}_E(\beta)(\alpha)(f) = f(\beta) \quad \text{for } \beta \in Obj \supseteq AObj, \\ &\bar{F}_E((e, \varphi))(\alpha)(f) = \bar{F}_E(e)(\alpha)(\lambda\beta \cdot \bar{F}_E(\varphi(\beta))(\alpha)(f)), \\ &\bar{F}_S(E)(\alpha)(p) = p, \\ &\bar{F}_S(\textbf{release}\ (\beta, s))(\alpha)(p) = p \parallel \bar{F}_S(s)(\beta)(p_0), \\ &\bar{F}_S((e, \psi))(\alpha)(p) = \bar{F}_E(e)(\alpha)(\lambda\beta \cdot \bar{F}_S(\psi(\beta))(\alpha)(p)).\end{aligned}$$

Finally, we set

$$\begin{aligned}\mathcal{D}'_U &= \langle \mathcal{D}'_E, \mathcal{D}'_S\rangle \\ &= \text{Fixed point } (\Psi'_U).\end{aligned}$$

The meaning of (e, φ) is obtained by first evaluating the expression e, then substituting the result β into the parameterized expression φ and finally evaluating the expression $\varphi(\beta)$. The interpretation of $\mathcal{D}'_S[\![(e, \psi)]\!]$ is similar. In $\mathcal{D}'_S[\![\textbf{release}\ (\beta, s)]\!](\alpha)(p)$, the meaning of the statement s (when executed by the object β and with the empty continuation p_0) is computed and composed in parallel with the process p, the continuation of the release statement.

DEFINITION 7.5 ($\mathcal{D}^*_U$). Let $\mathcal{D}^*_U : \mathcal{P}_{fin}(LStat) \to \bar{P}$ be given by

$$\mathcal{D}^*_U = (\tilde{\mathcal{D}}'_U),$$

where $\sim : (Q_E \times Q_S) \to (\mathcal{P}_{fin}(LStat) \to \bar{P})$ is defined as follows. If $F = \langle F_E, F_S \rangle$, then $\sim(F)$, here being denoted by $\tilde{F}$ is given by

$$\tilde{F}(\{(\alpha_1, s_1), \cdots, (\alpha_k, s_k)\}) = F_S(s_1)(\alpha_1)(p_0) \parallel \cdots \parallel F_S(s_k)(\alpha_k)(p_0).$$

(We put $\tilde{F}(\varnothing) = p_0$.)

Note that we have that $\tilde{F}(X \cup Y) = \tilde{F}(X) \parallel \tilde{F}(Y)$.

The omission of parentheses in the parallel composition above is justified by the fact that $\parallel$ is associative.

Given a finite set X of labelled statements (α_i, s_i), the value of $\mathcal{D}^*_U(X)$ is obtained by first computing the semantics of every labelled statement $(\alpha_i, s_i) \in X$. This is given by $\mathcal{D}_S[\![s_i]\!](\alpha_i)(p_0)$, where the label α_i indicates the name of the object executing the statement and where p_0 indicates that after s_i nothing remains to be done. Next, all the resulting processes are composed in parallel.

Now that we have extended the domain of $\mathcal{D}_U$ to $\mathcal{P}_{fin}(LStat)$ we are ready to prove the fact that $\mathcal{D}^*_U = \mathcal{O}'_U$. It is a straightforward corollary of Theorem 7.7 below. The proof of this theorem makes use of the following lemma.

LEMMA 7.6. *For all $\alpha \in AObj$ and $\psi \in L_{PS}$ we have*

$$\forall \beta[\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, \psi(\beta))\}) = \mathcal{D}^*_U(\{(\alpha, \psi(\beta))\})] \Rightarrow$$
$$\forall e \in L'_E[\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, (e, \psi))\}) = \mathcal{D}^*_U(\{(\alpha, (e, \psi))\})].$$

Proof. The proof uses induction on the complexity of expressions. We treat two simple basic cases, being (lazy and) confident that these will show the reader how to proceed in the other cases. So let $\alpha \in AObj$ and $\psi \in L_{PS}$ and suppose that

$$\forall \beta[\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, \psi(\beta))\}) = \mathcal{D}^*_U(\{(\alpha, \psi(\beta))\})].$$

For $e = \beta$ we have

$$\begin{aligned}
\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, (\beta, \psi))\}) &= \Phi'_U(\mathcal{D}^*_U)(\{(\alpha, \psi(\beta))\}) \\
&= [\text{hypothesis}] \\
&\quad \mathcal{D}^*_U(\{(\alpha, \psi(\beta))\}) \\
&= \mathcal{D}'_S[\![\psi(\beta)]\!](\alpha)(p_0) \\
&= \mathcal{D}'_S[\![(\beta, \psi)]\!](\alpha)(p_0) \\
&= \mathcal{D}^*_U(\{(\alpha, (\beta, \psi))\});
\end{aligned}$$

if $e = \beta_1 ! m(\beta_2)$ then

$$\begin{aligned}
\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, (\beta_1 ! m(\beta_2), \psi))\}) &= \lambda\sigma \cdot \{\langle \beta_1, m, \beta_2, \lambda\beta \cdot \mathcal{D}^*_U(\{(\alpha, \psi(\beta))\}), p_0 \rangle\} \\
&= \lambda\sigma \cdot \{\langle \beta_1, m, \beta_2, \lambda\beta \cdot \mathcal{D}'_S[\![\psi(\beta)]\!](\alpha)(p_0), p_0 \rangle\} \\
&= \mathcal{D}'_E[\![\beta_1]\!](\alpha)(\lambda\beta'_1 \cdot \mathcal{D}'_E[\![\beta_2]\!](\alpha) \\
&\qquad (\lambda\beta'_1 \cdot \lambda\sigma \cdot \{\langle \beta'_1, m, \beta'_2, \lambda\beta \cdot \mathcal{D}'_S[\![\psi(\beta)]\!](\alpha) \\
&\qquad (p_0), p_0 \rangle\})) \\
&= \mathcal{D}'_E[\![\beta_1 ! m(\beta_2)]\!](\alpha)(\lambda\beta \cdot \mathcal{D}'_S[\![\psi(\beta)]\!](\alpha)(p_0)) \\
&= \mathcal{D}'_S[\![(\beta_1 ! m(\beta_2), \psi)]\!](\alpha)(p_0) \\
&= \mathcal{D}^*_U(\{(\alpha, (\beta_1 ! m(\beta_2), \psi))\}).
\end{aligned}$$

THEOREM 7.7. $\Phi'_U(\mathcal{D}^*_U) = \mathcal{D}^*_U$.

Proof. We show: $\forall X \in \mathcal{P}_{fin}(LStat)\ [\Phi'_U(\mathcal{D}^*_U)(X) = \mathcal{D}^*_U(X)]$, using induction on the number of elements in X.

Case 1. $X = \{(\alpha, s)\}$, with $\alpha \in AObj$, $s \in L'_S$.

The proof uses induction on the complexity of the statement s. We treat some typical cases.

(i) **answer** m:

$$\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, \textbf{answer } m)\}) = \lambda\sigma \cdot \{\langle \alpha, m, g_m\rangle\}$$

with

$$\begin{aligned} g_m &= \lambda\beta \cdot \lambda f \cdot \lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta' \cdot \lambda\hat{\sigma} \cdot \{\langle \hat{\sigma}', f(\beta') \parallel \mathcal{D}^*_U(\{(\alpha, E)\})\rangle\})\rangle\}\} \\ &= \lambda\beta \cdot \lambda f \cdot \lambda\bar{\sigma} \cdot \{\langle \bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta' \cdot \lambda\hat{\sigma} \cdot \{\langle \hat{\sigma}', f(\beta')\rangle\})\rangle\}. \end{aligned}$$

(and $\bar{\sigma}'$ and $\hat{\sigma}'$ as in Definition 6.1). If we compare this to the definition of $\mathcal{D}_S[\![\textbf{answer } m]\!]$ (Definition 5.3(S3)) we see

$$\begin{aligned} \lambda\sigma \cdot \{\langle \alpha, m, g_m\rangle\} &= \mathcal{D}_S[\![\textbf{answer } m]\!](\alpha)(p_0) \\ &= \mathcal{D}^*_U(\{(\alpha, \textbf{answer } m)\}). \end{aligned}$$

(ii) $x \leftarrow e$: we distinguish two subcases. First, if $e = \beta$, then

$$\begin{aligned} \Phi'_U(\mathcal{D}^*_U)(\{(\alpha, x \leftarrow \beta)\}) &= \lambda\sigma \cdot \{\langle \sigma\{\beta/\alpha, x\}, p_0\rangle\} \\ &= \mathcal{D}'_E[\![\beta]\!](\alpha)(\lambda\beta \cdot \lambda\sigma \cdot \{\langle \sigma\{\beta/\alpha, x\}, p_0\rangle\}) \\ &= \mathcal{D}'_S[\![x \leftarrow \beta]\!](\alpha)(p_0) \\ &= \mathcal{D}^*_U(\{(\alpha, x \leftarrow \beta)\}). \end{aligned}$$

If $e \notin Obj$, then

$$\begin{aligned} \Phi'_U(\mathcal{D}^*_U)(\{(\alpha, x \leftarrow e)\}) &= [\text{definition } -U \rightarrow] \\ &= \Phi'_U(\mathcal{D}^*_U)(\{(\alpha, (e, \lambda u \cdot x \leftarrow u))\}) \\ &= [\text{see (v) below}] \\ &\quad \mathcal{D}^*_U(\{(\alpha, (e, \lambda u \cdot x \leftarrow u))\}) \\ &= \mathcal{D}'_E[\![e]\!](\alpha)(\lambda\beta \cdot \mathcal{D}'_S[\![x \leftarrow \beta]\!](\alpha)(p_0)) \\ &= \mathcal{D}'_S[\![x \leftarrow e]\!](\alpha)(p_0) \\ &= \mathcal{D}^*_U(\{(\alpha, x \leftarrow e)\}). \end{aligned}$$

(iii) $s_1 ; s_2$: case analysis for s_1.

(iv) **do** e **then** s **od**:

$$\begin{aligned} &\Phi'_U(\mathcal{D}^*_U)(\{(\alpha, \textbf{do } e \textbf{ then } s \textbf{ od})\}) \\ &= \lambda\sigma \cdot \{\langle \sigma, \mathcal{D}^*_U(\{(\alpha, \textbf{if } e \textbf{ then } s \text{ ; } (\textbf{do } e \textbf{ then } s \textbf{ od}) \textbf{ else } E \textbf{ fi})\})\rangle\} \\ &= \lambda\sigma \cdot \{\langle \sigma, \mathcal{D}'_E[\![e]\!](\alpha)(\lambda\beta \cdot \text{if } \beta = tt \text{ then} \\ &\qquad\qquad \mathcal{D}'_S[\![s]\!](\alpha)(\mathcal{D}'_S[\![\textbf{do } e \textbf{ then } s \textbf{ od}]\!](\alpha)(p_0)) \text{ else } p_0 \text{ fi})\rangle\} \\ &= \mathcal{D}'_S[\![\textbf{do } e \textbf{ then } s \textbf{ od}]\!](\alpha)(p_0) \\ &= \mathcal{D}^*_U(\{(\alpha, \textbf{do } e \textbf{ then } s \textbf{ od})\}). \end{aligned}$$

(v) (e, ψ): by induction we have that the theorem holds for $(\alpha, \psi(\beta))$, for every $\beta \in Obj$. Now we can apply Lemma 7.6.

Case 2. $X \in \mathcal{P}_{fin}(LStat)$ and X has at least two elements. Suppose we have two disjoint sets X_1 and X_2 in $\mathcal{P}_{fin}(LStat)$ with $X = X_1 \cup X_2$ such that

$$\Phi'_U(\mathcal{D}^*_U)(X_i) = \mathcal{D}^*_U(X_i)$$

for $i = 1, 2$. Assume $X_1, X_2 \neq \{\langle \alpha_1, E\rangle, \cdots, \langle \alpha_n, E\rangle\}$. We shall show that from this induction hypothesis it follows that

$$\Phi'_U(\mathcal{D}^*_U)(X_1 \cup X_2) = \mathcal{D}^*_U(X_1 \cup X_2).$$

(This is proved in very much the same way as the fact that $\Phi'(\tilde{\mathcal{D}})(\rho) = \tilde{\mathcal{D}}(\rho)$ and $\Phi'(\tilde{\mathcal{D}})(\pi) = \tilde{\mathcal{D}}(\pi)$ implies $\Phi'(\tilde{\mathcal{D}})(\rho^\wedge \pi) = \tilde{\mathcal{D}}(\rho^\wedge \pi)$, which occurs in Theorem 2.14 of § 2.)

From the definition of $-U \rightarrow$ (Definition 4.8, (R10) and (R11)) it follows that

$$\Phi'_U(\mathcal{D}^*_U)(X_1 \cup X_2) = \lambda\sigma \cdot (X_1^\sigma \cup X_2^\sigma \cup Z).$$

Here

$$\begin{aligned} X_1^\sigma = \{&\langle \sigma', \mathcal{D}^*_U(X'_1 \cup X_2)\rangle : \langle X_1, \sigma\rangle - U, \tau \rightarrow \langle X'_1, \sigma'\rangle\} \\ &\cup \{\langle \beta_1, m, \beta_2, \lambda\beta \cdot \mathcal{D}^*_U(\{(\alpha, \psi(\beta))\}), \mathcal{D}^*_U(X'_1 \cup X_2)\rangle : \\ &\qquad \langle X_1, \sigma\rangle - U, (\alpha, \beta_1 ! m(\beta_2)) \rightarrow \langle X'_1 \cup \{(\alpha, \psi)\}, \sigma\rangle\} \\ &\cup \{\langle \alpha, m, g_m\rangle : \langle X_1, \sigma\rangle - U, (\alpha ? m) \rightarrow \langle X'_1 \cup \{(\alpha, s)\}, \sigma\rangle\} \end{aligned}$$

with

$$\begin{aligned} g_m = \lambda\beta \cdot \lambda f \cdot (&\lambda\sigma \cdot \{\langle \bar{\sigma}', \mathcal{D}_E[\![e_m]\!](\alpha)(\lambda\beta' \cdot \lambda\hat{\sigma} \\ &\cdot \{\langle \hat{\sigma}', f(\beta') \parallel \mathcal{D}^*_U(\{(\alpha, s)\})\rangle\})\} \parallel \mathcal{D}^*_U(X'_1 \cup X_2)) \end{aligned}$$

and e_m, $\bar{\sigma}'$ and $\hat{\sigma}'$ as in Definition 6.1. The set X_2^σ is like X_1^σ but with the roles of X_1 and X_2 interchanged. Finally,

$$\begin{aligned} Z = \{&\langle \sigma', \mathcal{D}^*_U(\{(\beta_1, (e_m, \lambda u \cdot (u_m \leftarrow \sigma_2(\beta_1)(u_m)\ ;\ \textbf{release}\ (\alpha, \psi(u))\ ;\ s)))\} \bigcup X'_1 \cup X'_2)\rangle : \\ &\langle X_i, \sigma\rangle - U, (\alpha, \beta_1 ! m(\beta_2)) \rightarrow \langle \{(\alpha, \psi)\} \cup X'_i, \sigma\rangle \quad \text{and} \\ &\langle X_j, \sigma\rangle - U, (\beta_1 ? m) \rightarrow \langle \{(\beta_1, s)\} \cup X'_j, \sigma\rangle, \quad \text{for } i = 1, j = 2 \text{ or } i = 2, j = 1\} \end{aligned}$$

(and $\sigma' = \sigma\{\beta_2/\beta_1, u_m\}$, $m \Leftarrow \langle u_m, e_m\rangle \in U$). The steps in X_i^σ correspond to the transition steps that can be made from $X_1 \cup X_2$ as a result of a transition step from X_i (by an application of (R10) in the definition of $-U \rightarrow$), for $i = 1, 2$.

The set Z contains those steps that correspond with a communication transition from $X_1 \cup X_2$, which results from a send transition from X_i and an answer transition from X_j (for $i = 1, j = 2$, or $i = 2, j = 1$) by an application of (R11).

Now we have

$$\begin{aligned} X_1^\sigma &= \Phi'_U(\mathcal{D}^*_U)(X_1)(\sigma) \between \mathcal{D}^*_U(X_2), \\ X_2^\sigma &= \Phi'_U(\mathcal{D}^*_U)(X_2)(\sigma) \between \mathcal{D}^*_U(X_1), \\ Z &= \Phi'_U(\mathcal{D}^*_U)(X_1)(\sigma) \mid_\sigma \Phi'_U(\mathcal{D}^*_U)(X_2)(\sigma). \end{aligned}$$

The proofs of these facts are not difficult (but tiresome and therefore omitted). It

follows that

$$\begin{aligned}
\Phi'_U(\mathcal{D}^*_U)(X_1 \cup X_2) &= \lambda\sigma \cdot (X_1^\sigma \cup X_2^\sigma \cup Z) \\
&= \lambda\sigma \cdot (\Phi'_U(\mathcal{D}^*_U)(X_1)(\sigma) \mathbin{\|\!\!\llcorner} \mathcal{D}^*_U(X_2) \\
&\qquad \cup \Phi'_U(\mathcal{D}^*_U)(X_2)(\sigma) \mathbin{\|\!\!\llcorner} \mathcal{D}^*_U(X_1) \\
&\qquad \cup \Phi'_U(\mathcal{D}^*_U)(X_2)(\sigma) \mid_\sigma \Phi'_U(\mathcal{D}^*_U)(X_2)(\sigma)) \\
&= [\text{induction hypothesis}] \\
&\quad \lambda\sigma \cdot (\mathcal{D}^*_U(X_1)(\sigma) \mathbin{\|\!\!\llcorner} \mathcal{D}^*_U(X_2) \\
&\qquad \cup \mathcal{D}^*_U(X_2)(\sigma) \mathbin{\|\!\!\llcorner} \mathcal{D}^*_U(X_1) \\
&\qquad \cup \mathcal{D}^*_U(X_1)(\sigma) \mid_\sigma \mathcal{D}^*_U(X_2)(\sigma)) \\
&= [\text{definition } \|] \ \mathcal{D}^*_U(X_1) \parallel \mathcal{D}^*_U(X_2) \\
&= \mathcal{D}^*_U(X_1 \cup X_2).
\end{aligned}$$

This concludes the proof of Theorem 7.7. □

Since $\mathcal{O}'_U$ and $\mathcal{D}^*_U$ are both fixed points of the same contraction Φ'_U, they must be equal.

COROLLARY 7.8. $\mathcal{O}'_U = \mathcal{D}^*_U$.

7.3. Collecting the results. We have proved that $\mathcal{O}_U = \mathit{abstr} \circ \mathcal{O}'_U$ and that $\mathcal{O}'_U = \mathcal{D}^*_U$. Thus we have the following theorem.

THEOREM 7.9. $\mathcal{O}_U = \mathit{abstr} \circ \mathcal{D}^*_U$.

From this theorem we deduce the main theorem of this paper.

THEOREM 7.10. $[\![U]\!]_{\mathcal{O}} = \mathit{abstr}([\![U]\!]_{\mathcal{D}})$.

Proof. Let $U = \langle (\cdots, C_n \Leftarrow s_n), \cdots \rangle$. Then

$$\begin{aligned}
[\![U]\!]_{\mathcal{O}} &= \mathcal{O}_U[\![\{(\nu(\varnothing), s_n)\}]\!] \\
&= \mathit{abstr}(\mathcal{D}^*_U(\{(\nu(\varnothing), s_n)\})) \\
&= \mathit{abstr}(\mathcal{D}'_S[\![s_n]\!](\nu(\varnothing))(p_0)) \\
&= \mathit{abstr}(\mathcal{D}_S[\![s_n]\!](\nu(\varnothing))(p_0)) \\
&= \mathit{abstr}([\![U]\!]_{\mathcal{D}}).
\end{aligned}$$

Appendix I. Mathematical definitions.

DEFINITION I.1 (Metric space). A *metric space* is a pair (M, d) with M a nonempty set and d a mapping $d : M \times M \to [0, 1]$ (a *metric* or *distance*) that satisfies the following properties:

(a) $\forall x, y \in M[d(x, y) = 0 \Leftrightarrow x = y]$,

(b) $\forall x, y \in M[d(x, y) = d(y, x)]$,

(c) $\forall x, y, z \in M[d(x, y) \leqq d(x, z) + d(z, y)]$.

We call (M, d) an *ultra-metric space* if the following stronger version of property (c) is satisfied:

(c') $\forall x, y, z \in M \ [d(x, y) \leqq \max\{d(x, z), d(z, y)\}]$.

Please note that we consider only metric spaces with bounded diameter: the distance between two points never exceeds one.

Examples I.1.1. (a) Let A be an arbitrary set. The *discrete* metric d_A on A is defined as follows. Let $x, y \in A$, then

$$d_A(x, y) = \begin{cases} 0 & \text{if } x = y, \\ 1 & \text{if } x \neq y. \end{cases}$$

(b) Let A be an alphabet, and let $A^\infty = A^* \cup A^\omega$ denote the set of all finite and infinite words over A. Let, for $x \in A^\infty$, $x[n]$ denote the prefix of x of length n, in case $\mathit{length}(x) \geqq n$, and x otherwise. We put

$$d(x, y) = 2^{-\sup\{n : x[n] = y[n]\}}$$

with the convention that $2^{-\infty} = 0$. Then (A^∞, d) is a metric space.

DEFINITION I.2. Let (M, d) be a metric space, let $(x_i)_i$ be a sequence in M.

(a) We say that $(x_i)_i$ is a *Cauchy sequence* whenever we have: $\forall \varepsilon > 0\ \exists N \in \mathbb{N}\ \forall n, m > N[d(x_n, x_m) < \varepsilon]$.

(b) Let $x \in M$. We say that $(x_i)_i$ *converges to* x and call x the *limit* of $(x_i)_i$ whenever we have: $\forall \varepsilon > 0\ \exists n \in \mathbb{N}\ \forall n > N[d(x, x_n) < \varepsilon]$. Such a sequence we call *convergent.* Notation: $\lim_{i \to \infty} x_i = x$.

(c) The metric space (M, d) is called complete whenever each Cauchy sequence converges to an element of M.

DEFINITION I.3. Let (M_1, d_1), (M_2, d_2) be metric spaces.

(a) We say that (M_1, d_1) and (M_2, d_2) are *isometric* if there exists a bijection $f : M_1 \to M_2$ such that: $\forall x, y \in M_1[d_2(f(x), f(y)) = d_1(x, y)]$. We then write $M_1 \cong M_2$. When f is not a bijection (but only an injection), we call it an *isometric embedding.*

(b) Let $f : M_1 \to M_2$ be a function. We call f *continuous* whenever for each sequence $(x_i)_i$ with limit x in M_1 we have that $\lim_{i \to \infty} f(x_i) = f(x)$.

(c) Let $A \geqq 0$. With $M_1 \to^A M_2$ we denote the set of functions f from M_1 to M_2 that satisfy the following property: $\forall x, y \in M_1[d_2(f(x), f(y)) \leqq A \cdot d_1(x, y)]$. Functions f in $M_1 \to^1 M_2$ we call *nonexpansive*, functions f in $M_1 \to^\varepsilon M_2$ with $0 \leqq \varepsilon < 1$ we call *contracting.* (For every $A \geqq 0$ and $f \in M_1 \to^A M_2$ we have that f is continuous.)

PROPOSITION I.4 (Banach's fixed-point theorem). *Let (M, d) be a complete metric space and $f : M \to M$ a contracting function. Then there exists an $x \in M$ such that the following holds*:

(1) $f(x) = x$ (*x is a fixed point of f*),

(2) $\forall y \in M[f(y) = y \Rightarrow y = x]$ (*x is unique*),

(3) $\forall x_0 \in M[\lim_{n \to \infty} f^{(n)}(x_0) = x]$, *where* $f^{(n+1)}(x_0) = f(f^{(n)}(x_0))$ *and* $f^{(0)}(x_0) = x_0$.

DEFINITION I.5 (closed and compact subsets). A subset X of a complete metric space (M, d) is called *closed* whenever each Cauchy sequence in X has a limit in X and is called *compact* whenever each sequence in X has a subsequence that converges to an element of X.

DEFINITION I.6. Let $(M, d), (M_1, d_1), \cdots, (M_n, d_n)$ be metric spaces.

(a) With $M_1 \to M_2$ we denote the set of all continuous functions from M_1 to M_2. We define a metric d_F on $M_1 \to M_2$ as follows. For every $f_1, f_2 \in M_1 \to M_2$

$$d_F(f_1, f_2) = \sup_{x \in M_1} \{d_2(f_1(x), f_2(x))\}.$$

For $A \geqq 0$ the set $M_1 \to^A M_2$ is a subset of $M_1 \to M_2$, and a metric on $M_1 \to^A M_2$ can be obtained by taking the restriction of the corresponding d_F.

(b) With $M_1 \mathbin{\overline{\cup}} \cdots \mathbin{\overline{\cup}} M_n$ we denote the *disjoint union* of $M_1, \cdots, M_n$, which can be defined as $\{1\} \times M_1 \cup \cdots \cup \{n\} \times M_n$. We define a metric d_U on $M_1 \mathbin{\overline{\cup}} \cdots \mathbin{\overline{\cup}} M_n$ as

follows. For every $x, y \in M_1 \cup \cdots \cup M_n$

$$d_U(x, y) = \begin{cases} d_j(x, y) & \text{if } x, y \in \{j\} \times M_j, \quad 1 \leqq j \leqq n, \\ 1 & \text{otherwise.} \end{cases}$$

(c) We define a metric d_P on $M_1 \times \cdots \times M_n$ by the following clause. For every $(x_1, \cdots, x_n), (y_1, \cdots, y_n) \in M_1 \times \cdots \times M_n$

$$d_P((x_1, \cdots, x_n), (y_1, \cdots, y_n)) = \max_i \{d_i(x_i, y_i)\}.$$

(d) Let $\mathcal{P}_{closed}(M) = \{X: X \subseteq M \wedge X \text{ is closed}\}$. We define a metric d_H on $\mathcal{P}_{closed}(M)$, called the *Hausdorff distance*, as follows. For every $X, Y \in \mathcal{P}_{closed}(M)$ with $X, Y \neq \varnothing$

$$d_H(X, Y) = \max \left\{ \sup_{x \in X} \{d(x, Y)\}, \sup_{y \in Y} \{d(y, X)\} \right\},$$

where $d(x, Z) =^{def} \inf_{z \in Z} \{d(x, z)\}$ for every $Z \subseteq M$, $x \in M$. For $X \neq \varnothing$ we put

$$d_H(\varnothing, X) = d_H(X, \varnothing) = 1.$$

The following spaces:

$$\mathcal{P}_{compact}(M) = \{X: X \subseteq M \wedge X \text{ is compact}\},$$

$$\mathcal{P}_{ncompact}(M) = \{X: X \subseteq M \wedge X \text{ is nonempty and compact}\}$$

are supplied with a metric by taking the respective restrictions of d_H.

(e) Let $c \in [0, 1]$. We define: $id_c(M, d) = (M, c \cdot d)$.

PROPOSITION I.7. *Let* (M, d), $(M_1, d_1), \cdots, (M_n, d_n)$, d_F, d_U, d_P, *and* d_H *be as in Definition* I.6 *and suppose that* (M, d), $(M_1, d_1), \cdots, (M_n, d_n)$ *are complete. We have that*

(a) $(M_1 \to M_2, d_F)$, $(M_1 \to^A M_2, d_F)$,

(b) $(M_1 \cup \cdots \cup M_n, d_U)$,

(c) $(M_1 \times \cdots \times M_n, d_P)$,

(d) $(\mathcal{P}_{closed}(M), d_H)$, $(\mathcal{P}_{compact}(M), d_H)$ and $(\mathcal{P}_{ncompact}(M), d_H)$ *are complete metric spaces. If* (M, d) *and* (M_i, d_i) *are all ultra-metric spaces these composed spaces are again ultra-metric. (Strictly speaking, for the completeness of* $M_1 \to M_2$ *and* $M_1 \to^A M_2$ *we do not need the completeness of* M_1. *The same holds for the ultra-metric property.)*

The proofs of Proposition I.7(a)–(c) are straightforward. Part (d) is more involved. It can be proved with the help of the following characterization of the completeness of the Hausdorff metric.

PROPOSITION I.8. *Let* $(\mathcal{P}_{closed}(M), d_H)$ *be as in Definition* I.6. *Let* $(X_i)_i$ *be a Cauchy sequence in* $\mathcal{P}_{closed}(M)$. *We have*

$$\lim_{i \to \infty} X_i = \{\lim_{i \to \infty} x_i \mid x_i \in X_i, (x_i)_i \text{ a Cauchy sequence in } M\}.$$

The proof of Proposition I.8 can be found in [Du66] and [En77]. The completeness of the Hausdorff space containing compact sets is proved in [Mi51].

Appendix II. The function *abstr*. The definition of $abstr: \bar{P} \to P$ can be viewed as a fixed-point characterization of a somewhat differently and more intuitively defined operation

$$abstr^*: \bar{P} \to P,$$

which we introduce below. Next, we show that $abstr = abstr^*$.

DEFINITION II.1 (*abstr**). Let $p \in \bar{P}$ and $\sigma \in \Sigma$, and let $w \in \Sigma_\partial^\infty$.

(1) We call w a *finite stream* in $p(\sigma)$ if there exist $\langle \sigma_1, p_1 \rangle, \cdots, \langle \sigma_n, p_n \rangle$ such that

$$w = \sigma_1 \cdots \sigma_n \wedge \forall 1 \leqq i < n[\langle \sigma_{i+1}, p_{i+1} \rangle \in p_i(\sigma_i)] \wedge \langle \sigma_1, p_1 \rangle \in p(\sigma) \wedge p_n = p_0.$$

(2) We call w an *infinite stream* in $p(\sigma)$ if there exist $\langle \sigma_1, p_1 \rangle, \langle \sigma_2, p_2 \rangle, \cdots$ such that

$$w = \sigma_1 \sigma_2 \cdots \wedge \forall 1 \leqq i[\langle \sigma_{i+1}, p_{i+1} \rangle \in p_i(\sigma)] \wedge \langle \sigma_1, p_1 \rangle \in p(\sigma).$$

(3) We call w a *deadlocking stream* in $p(\sigma)$ if there exist $\langle \sigma_1, p_1 \rangle, \cdots, \langle \sigma_n, p_n \rangle$ such that

$$w = \sigma_1 \cdots \sigma_n \cdot \partial \wedge \forall 1 \leqq i < n[\langle \sigma_{i+1}, p_{i+1} \rangle \in p_i(\sigma_i)]$$
$$\wedge \langle \sigma_1, p_1 \rangle \in p(\sigma) \wedge p_n \neq p_0 \wedge p_n(\sigma_n) \cap (\Sigma \times \bar{P}) = \varnothing.$$

Now we define a function $abstr^* : \bar{P} \to P$ by

$$abstr^*(p) = \lambda\sigma \cdot \{w : w \text{ is a stream in } p(\sigma)\}.$$

We have to verify that for every $p \in \bar{P}$ and $\sigma \in \Sigma$ the set $abstr^*(p)(\sigma)$ is compact. This is not trivial and is proved in Theorem II.3 below (which is a slightly generalised form of Lemma AII.4 in [BBKM84]). The fact that we use in the definition of $\bar{P}$ *compact* subsets rather than closed ones is essential for the proof. (For a process domain defined with *closed* subsets, [BBKM84] provides a counterexample of the theorem.)

In the proof of Theorem II.3 below, we need the following lemma.

LEMMA II.2. *Let* $q = \lim_{n\to\infty} q_n$, *for* $q, q_n \in \bar{P}$: *assume* (*without loss of generality*) *that for all* $n \geqq 0$

$$d(q, q_n) \leqq 2^{-(n+1)}$$

Let $\sigma \in \Sigma$ *and let* $(w_i)_i$ *be a sequence in* Σ_∂^∞ *with* $w_i \in abstr^*(q_i)(\sigma)$, *for every* $i \geqq 0$. *Then*

$$\forall n\ \exists u\ [w_n[n] \cdot u \in abstr^*(q)(\sigma)].$$

Proof. Let $w_n[n] = \sigma_1 \cdots \sigma_n$. (In the case of termination or deadlock the rest of the proof is analogous to this case.) Now there must be $q^1, \cdots, q^n$ with

$$\langle \sigma_1, q^1 \rangle \in q_n(\sigma) \quad \text{and} \quad \langle \sigma_{i+1}, q^{i+1} \rangle \in q^i(\sigma_i)$$

for $1 \leqq i \leqq n$. We shall show that there are $\bar{q}^1, \cdots, \bar{q}^n$ with $\langle \sigma_1, \bar{q}^1 \rangle \in q(\sigma)$ and $\langle \sigma_{i+1}, \bar{q}^{i+1} \rangle \in \bar{q}^i(\sigma_i)$ for $1 \leqq i \leqq n$. We do this inductively as follows. For $i = 1$ we observe that $d(q, q_n) \leqq 2^{-(n+1)}$, so $d(q(\sigma), q_n(\sigma)) \leqq 2^{-n} \leqq \frac{1}{2}$. Because $\langle \sigma_1, q^1 \rangle \in q_n(\sigma)$, there must be a $\bar{q}^1$ with

$$\langle \sigma_1, \bar{q}^1 \rangle \in q(\sigma) \quad \text{and} \quad d(q^1, \bar{q}^1) \leqq 2^{-n}.$$

For the inductive step, let $1 \leqq i \leqq n$ and let $\bar{q}^i$ be such that $d(q^i, \bar{q}^i) \leqq 2^{-(n+1)+i}$. Then

$$d(q^i(\sigma_i), \bar{q}^i(\sigma_i)) \leqq 2^{-n+i} \leqq \tfrac{1}{2}.$$

Because $\langle \sigma_{i+1}, q^{i+1} \rangle \in q^i(\sigma_i)$ there must be a $\bar{q}^{i+1}$ with

$$\langle \sigma_{i+1}, \bar{q}^{i+1} \rangle \in \bar{q}^i(\sigma_i) \quad \text{and} \quad d(q^{i+1}, \bar{q}^{i+1}) \leqq 2^{-n+i}.$$

With $\bar{q}^1, \cdots, \bar{q}^n$ suitably chosen, we can take $u \in abstr^*(\bar{q}^n)(\sigma_n)$ arbitrary, and then $w_n[n] \cdot u$ will be in $abstr^*(q)(\sigma)$.

THEOREM II.3. *For every $p \in \bar{P}$ and $\sigma \in \Sigma$ the set $abstr^*(p)(\sigma)$ is compact.*

Proof. Let $(w_i)_i$ be a sequence in $abstr^*(p)(\sigma)$. We shall show that there exists a subsequence of $(w_i)_i$ that has its limit in $abstr^*(p)(\sigma)$. First we introduce some notation. For an arbitrary word $w \in \Sigma_\partial^\infty$, $w\langle k\rangle$ indicates the word that is obtained from w by omitting the first k elements. We call $p_0 = p$, $\sigma_0 = \sigma$, and $f_0 = id_{\mathbb{N}}$, the identity function on the set of natural numbers. We shall inductively construct for every $n \geqq 0$ a function $f_n : \mathbb{N} \to \mathbb{N}$, a process $p_n \in \bar{P}$, and a state σ_n such that

1. $\forall i \geqq 0[w_{f_n(i)}[n] = \sigma_1 \cdot \cdots \cdot \sigma_n]$.
2. $\forall i,\ 0 \leqq i < n[\langle \sigma_{i+1}, p_{i+1}\rangle \in p_i(\sigma_i)]$.
3. $\exists (\nu_i)_i$ in $abstr^*(p_n)(\sigma_n)$ $\forall i \geqq 1[\nu_i[i] = w_{f_n(i)}\langle n\rangle[i]]$.
4. f_n is monotonic and there exists a monotonic h with $f_n = f_{n-1} \circ h$.

Once we have constructed such sequences $(f_n)_n$, $(p_n)_n$, and $(\sigma_n)_n$, we are done. We can define

$$g(i) = f_i(i).$$

This function is monotonic and we have

$$\lim_{i \to \infty} w_{g(i)} = \sigma_1 \cdot \sigma_2 \cdot \cdots .$$

Since $\sigma_1 \cdot \sigma_2 \cdot \cdots \in abstr^*(p)(\sigma)$ we thus have found a subsequence $(w_{g(i)})_i$ of $(w_i)_i$, which has its limit in $abstr^*(p)(\sigma)$.

The construction is as follows. Suppose we are at stage $n \geqq 0$. Let $(\nu_i)_i$ be a sequence in $abstr^*(p_n)(\sigma_n)$ satisfying property 3, above. Let for every $i \geqq 1$

$$\nu_i = \tau_1^i \cdot \tau_2^i \cdot \cdots .$$

Then there are $q_1^i, q_2^i, \cdots \in \bar{P}$ with

$$\langle \tau_1^i, q_1^i\rangle \in p_n(\sigma_n) \quad \text{and} \quad \forall j \geqq 1[\langle \tau_{j+1}^i, q_{j+1}^i\rangle \in q_j^i(\tau_j^i)].$$

Since the set $p_n(\sigma_n)$ is compact, the sequence $(\langle \tau_1^i, q_1^i\rangle)_i$ has a converging subsequence that is given by, say, the monotonic function h and that has a limit, say $\langle \tau, q\rangle$ in $p_n(\sigma_n)$. We may assume

$$\forall j \geqq 1[\tau_1^{h(j)} = \tau \wedge d(q_1^{h(j)}, q) \leqq 2^{-(j+1)}].$$

Now we take

$$p_{n+1} = q, \quad \sigma_{n+1} = \tau, \quad f_{n+1} = f_n \circ h.$$

In order to show that this construction works, we must verify that p_{n+1}, σ_{n+1}, and f_{n+1} again satisfy properties 1–4 above.

1. We have for every $i \geqq 1$:

$$\begin{aligned} w_{f_{n+1}(i)}[n+1] &= w_{f_{n+1}(i)}[n] \cdot w_{f_{n+1}(i)}(n+1) \\ &= \sigma_1 \cdot \cdots \cdot \sigma_n \cdot w_{f_{n+1}(i)}\langle n\rangle(1) \\ &= \sigma_1 \cdot \cdots \cdot \sigma_n \cdot \nu_{h(i)}(1) \end{aligned}$$

$$= \sigma_1 \cdot \cdots \sigma_n \cdot \sigma_{n+1}.$$

2. We have $\langle \sigma_{n+1}, p_{n+1} \rangle = \langle \tau, q \rangle \in p_n(\sigma_n)$.

3. To prove this property, we are going to apply the following version of Lemma II.2. For all $q, q_1, q_2, \cdots \in \bar{P}$, and for all $x_1, x_2, \cdots \in \Sigma_\partial^\infty$,

$$\forall i \geqq 1[d(q, q_i) \leqq 2^{-(i+1)} \wedge x_i \in abstr^*(q_i)(\sigma)]$$
$$\Rightarrow \exists (\nu_i)_i \text{ in } abstr^*(q)(\sigma) \quad \forall i \geqq 1[y_i[i] = x_i[i]].$$

This we now use. Since

$$\forall i \geqq 1[d(p_{n+1}, q_1^{h(i)}) \leqq 2^{-(i+1)} \wedge \nu_{h(i)}\langle 1 \rangle \in abstr^*(q_1^{h(i)})(\sigma_{n+1})]$$

there must exist a sequence $(\nu_i')_i$ in $abstr^*(p_{n+1})(\sigma_{n+1})$ with

$$\forall i \geqq 1[\nu_i'[i] = \nu_{h(i)}\langle 1 \rangle[i]].$$

Now

$$\begin{aligned} \nu_{h(i)}\langle 1 \rangle[i] &= \nu_{h(i)}[h(i)]\langle 1 \rangle[i] \\ &= w_{f_{n+1}(i)}\langle n \rangle[h(i)]\langle 1 \rangle[i] \\ &= w_{f_{n+1}(i)}\langle n \rangle\langle 1 \rangle[i] \\ &= w_{f_{n+1}(i)}\langle n+1 \rangle[i]. \end{aligned}$$

(Here we have used twice the fact that $h(i) > i$, for all $i \geqq 1$.)

4. By definition.

This concludes the proof of Theorem II.3.

Next we show that the function $abstr: \bar{P} \to P$, given in Definition 7.1, can be defined as the fixed point of a contraction.

DEFINITION II.4 (Formal definition *abstr*). We define $\Xi: (\bar{P} \to^1 P) \to (\bar{P} \to^1 P)$; let $F \in P \to^1 P$, $P \in \bar{P}$, and $\sigma \in \Sigma$. We put

$$\Xi(F)(p_0)(\sigma) = \{\varepsilon\},$$
$$\Xi(F)(p)(\sigma) = \{\partial\} \quad \text{if } p(\sigma) \cap Comp_{\bar{P}} = \varnothing.$$

Otherwise, we set

$$\Xi(F)(p)(\sigma) = \bigcup \{\sigma' \cdot F(p')(\sigma') : \langle \sigma', p' \rangle \in p(\sigma)\}.$$

Finally, we define

$$abstr = \text{Fixed point } (\Xi).$$

It is straightforward to show Ξ is contracting. The fact that for every $p \in \bar{P}$ and $\sigma \in \Sigma$ the set $\Xi(F)(p)(\sigma)$ is compact needs some explanation. In order to prove this, it is convenient to adapt the definition of Ξ a little. Recalling that $P = \Sigma \to \mathcal{P}_{ncompact}(\Sigma_\partial^\infty)$, we define

$$\Xi': ((\bar{P} \times \Sigma) \to^1 \mathcal{P}_{ncompact}(\Sigma_\partial^\infty)) \to ((\bar{P} \times \Sigma) \to^1 \mathcal{P}_{ncompact}(\Sigma_\partial^\infty)),$$

where the superscript 1 above the arrow indicates that we consider only nonexpansive (and hence continuous) functions, by

$$\Xi'(F)(\langle p, \sigma \rangle) = \bigcup \{\sigma' \cdot F(\langle p', \sigma' \rangle) : \langle \sigma', p' \rangle \in p(\sigma)\}.$$

Now

$$\begin{aligned} \Xi'(F)(\langle p, \sigma \rangle) &= \bigcup_{\langle \sigma', p' \rangle \in p(\sigma)} \{\sigma' \cdot F(\langle p', \sigma' \rangle)\} \\ &= \bigcup_{\sigma'} (\sigma' \cdot \{F(\langle p', \sigma' \rangle) : \langle \sigma', p' \rangle \in p(\sigma)\}) \\ &= \bigcup_{\sigma'} (\sigma' \cdot F(\{\langle p', \sigma' \rangle : \langle \sigma', p' \rangle \in p(\sigma)\})). \end{aligned}$$

This union can be seen to be compact by first observing that from the compactness of p it follows that the union is finite: the set

$$\{\sigma' : \exists p' \in \bar{P}[\langle \sigma', p' \rangle \in p(\sigma)]\}$$

is finite. The compactness of $p(\sigma)$ further implies the compactness of the isomorphic set

$$\{\langle p', \sigma' \rangle : \langle \sigma', p' \rangle \in p(\sigma)\},$$

for every $\sigma' \in \Sigma$, which is preserved under the continuous mapping F and the concatenation with σ'. So we have a finite union of compact sets, which is again compact. Now the compactness of $\Xi(F)(p)(\sigma)$ follows straightforwardly from the compactness of $\Xi'(F')(\langle p', \sigma' \rangle)$, for arbitrary F', p', and σ'. The fact that $\Xi(F)$ is again nonexpansive is also easily verified.

We conclude this Appendix by showing that *abstr* and *abstr** are equal.

THEOREM II.5. *abstr* = *abstr**.

Proof. Consider $p \in \bar{P} - \{p_0\}$ and $\sigma \in \Sigma$ such that $p(\sigma) \cap (\Sigma \times \bar{P}) \neq \emptyset$. Then

$$\begin{aligned} w \in \mathit{abstr}^*(p)(\sigma) &\Leftrightarrow [\text{definition } \mathit{abstr}^*] \\ &\quad \exists \sigma' \in \Sigma \quad \exists w' \in \Sigma_\partial^\infty \quad \exists p' \in \bar{P}[w = \sigma' \cdot w' \wedge w' \in \mathit{abstr}^*(p')(\sigma')] \\ &\Leftrightarrow [\text{definition } \Xi] \\ &\quad w \in \Xi(\mathit{abstr})(p)(\sigma). \end{aligned}$$

The other cases are easy. We see: $\mathit{abstr}^* = \Xi(\mathit{abstr}^*)$. Because Ξ is a contraction the theorem follows. (Note the similarity of this proof and the one of Theorem 4.14.)

Appendix III. Standard objects. We want to extend the language under consideration with a few standard classes of so-called *standard* objects, namely, the classes `Boolean` and `Integer`. On these objects the usual operations can be performed, but they must be formulated by sending messages. For example, the addition 23+11 is indicated by the send expression 23!add(11), sending a message with method name add and parameter 11 to the standard object 23. The set of expressions L_E, given in Definition 3.1, is extended with these standard objects:

$$e ::= x | u | e_1 ! m(e_2) | \mathbf{new}(C) | s ; e | \mathbf{self} | \alpha,$$

where $\alpha \in \mathit{SObj}$, with

$$\mathit{SObj} = \mathbb{Z} \cup \{tt, ff\}.$$

Recall that we already defined (in Definition 4.1)

$$\begin{aligned} \mathit{Obj} &= \mathit{AObj} \cup \mathit{SObj} \\ &(= \mathit{AObj} \cup \mathbb{Z} \cup \{tt, ff\}). \end{aligned}$$

Intuitively, the evaluation of the expression α, with $\alpha \in \mathit{SObj}$, results in that object itself. For instance, the value of the expression 29 will be the integer 29.

Below, we shall first extend the definition of the operational semantics, next we adapt the definition of the denotational semantics (following [ABKR86(b)]), and finally we shall prove that the equivalence result of § 7 still holds.

III.1. Standard objects in the operational semantics. We extend the set $L_{E'}$, given in Definition 4.2, with the standard objects:

$$e ::= x | u | e_1 ! m(e_2) | \mathbf{new}(C) | s ; e | \mathbf{self} | \alpha | (e, \phi),$$

where now $\alpha \in \mathit{Obj} = \mathit{AObj} \cup \mathit{SObj}$.

Next we add to the set of labelled statements (Definition 4.5) an abstract element S_t that represents all standard objects and for which transitions will be specified in a moment:

$$LStat^* = LStat \cup \{S_t\}.$$

The following transitions are possible from S_t:

$$\langle\{S_t\}, \sigma\rangle - n\,?\mathsf{add} \to \langle\{S_t\}, \sigma\rangle, \qquad \langle\{S_t\}, \sigma\rangle - n\,?\mathsf{sub} \to \langle\{S_t\}, \sigma\rangle,$$

$$\langle\{S_t\}, \sigma\rangle - b\,?\mathsf{and} \to \langle\{S_t\}, \sigma\rangle, \qquad \langle\{S_t\}, \sigma\rangle - b\,?\mathsf{or} \to \langle\{S_t\}, \sigma\rangle,$$

$$\langle\{S_t\}, \sigma\rangle - b\,?\mathsf{not} \to \langle\{S_t\}, \sigma\rangle$$

for every $n \in \mathbb{Z}$ and $b \in \{tt, ff\}$. (This list can be extended with transitions for other operations.) Communication with a standard object is now modelled by the following transitions:

If $\langle\{(\alpha, s)\}, \sigma\rangle - (\alpha, n\,!\mathsf{add}\,(m)) \to \langle\{(\alpha, \psi)\}, \sigma\rangle$,
then $\langle\{(\alpha, s), S_t\}, \sigma\rangle - \gamma \to \langle\{(\alpha, \psi(n+m)), S_t\}$.
If $\langle\{(\alpha, s)\}, \sigma\rangle - (\alpha, b_1\,!\mathsf{and}\,(b_2)) \to \{(\alpha, \psi\}, \sigma\rangle$,
then $\langle\{(\alpha, s), S_t\}, \sigma\rangle - \gamma \to \langle\{(\alpha, \psi(b_1 \wedge b_2)), S_t\}, \sigma\rangle$,

and by similar transitions for the other operations. The result of, for example, an addition of the integers n and m is computed and passed through to the parameterized statement of the object requesting the execution of the method add.

Finally, the operational semantics of a unit (Definition 4.11) is changed by taking into account the standard objects; we put

$$[\![U]\!]_{\mathcal{O}} = \mathcal{O}_U[\![\{(\nu(\varnothing), s_n), S_t\}]\!].$$

(In the operational semantics defined in [ABKR86(a)], the standard objects are treated somewhat differently. There no special rules are given for the communication with a standard object; instead, some axioms are added that replace in one step a send expression that addresses a standard object by the corresponding value of the result.)

III.2. Standard objects in the denotational semantics. The denotational meaning of a standard object $\alpha \in L_E$ is given by

$$\mathcal{D}_E[\![\alpha]\!](\beta)(f) = f(\alpha),$$

where $\beta \in AObj$, and $f \in Cont_E$.

We follow [ABKR86(b)] in introducing a process $p_{St} \in \bar{P}$ that represents the denotational meaning of the standard objects. For this we have to adapt our semantic process domain $\bar{P}$. In Definition 5.1 the domain $\bar{P}$ is given by

$$\bar{P} \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{compact}(Step_{\bar{P}})).$$

In order to let the standard process p_{St}, to be defined below, fit into our semantic domain nicely, we are forced to use closed subsets of steps rather than compact ones. Let us indicate the process domain given in Definition 5.1 by $\bar{P}_{co}$. We introduce here $\bar{P}_{cl}$, which satisfies

$$\bar{P}_{cl} \cong \{p_0\} \cup id_{1/2}(\Sigma \to \mathcal{P}_{closed}(Step_{\bar{P}_{cl}})).$$

We have, via an obvious embedding, that $\bar{P}_{co} \subseteq \bar{P}_{cl}$.

Next we introduce $p_{St} \in \bar{P}_{cl}$, which represents the meaning of all standard objects. It satisfies the following equation:

$$\begin{aligned} p_{St} = \lambda\sigma \cdot (&\{\langle n, \mathsf{add}, g_n^+\rangle : n \in \mathbb{Z}\} \\ &\cup \{\langle n, \mathsf{sub}, g_n^-\rangle : n \in \mathbb{Z}\} \\ &\cup \{\langle b, \mathsf{and}, g_b^\wedge\rangle : b \in \{tt, ff\}\} \\ &\cup \{\langle b, \mathsf{or}, g_b^\vee\rangle : b \in \{tt, ff\}\} \\ &\cup \{\langle b, \mathsf{not}, g_b^\neg\rangle : b \in \{tt, ff\}\}), \end{aligned}$$

where

$$g_n^+ = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to \bar{P} \cdot (\text{if } \bar{\beta} \in \mathbb{Z} \text{ then } f(n + \bar{\beta}) \parallel p_{St} \text{ else } p_{St} \text{ fi}),$$

$$g_n^- = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to \bar{P} \cdot (\text{if } \bar{\beta} \in \mathbb{Z} \text{ then } f(n - \bar{\beta}) \parallel p_{St} \text{ else } p_{St} \text{ fi}),$$

$$g_b^\wedge = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to \bar{P} \cdot (\text{if } \bar{\beta} \in \{tt, ff\} \text{ then } f(b \wedge \bar{\beta}) \parallel p_{St} \text{ else } p_{St} \text{ fi}),$$

$$g_b^\vee = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to \bar{P} \cdot (\text{if } \bar{\beta} \in \{tt, ff\} \text{ then } f(b \vee \bar{\beta}) \parallel p_{St} \text{ else } p_{St} \text{ fi}),$$

$$g_b^\neg = \lambda\bar{\beta} \in Obj^* \cdot \lambda f \in Obj \to \bar{P} \cdot f(\neg b) \parallel p_{St}.$$

This definition is self-referential since p_{St} occurs at the right-hand side of the definition. Formally, p_{St} can be given as the fixed point of a suitably defined contraction on $\bar{P}_{cl}$.

We observe that p_{St} is an infinitely branching process, which is an element of $\bar{P}_{cl}$ but not of $\bar{P}_{co}$. This explains the introduction of $\bar{P}_{cl}$.

The operational intuition behind the definition of p_{St} is the following: For every $n \in \mathbb{Z}$ the set $p_{St}(\sigma)$ contains, among others, two elements, namely $\langle n, \mathsf{add}, g_n^+\rangle$ and $\langle n, \mathsf{sub}, g_n^-\rangle$. These steps indicate that the integer object n is willing to execute its methods add and sub. If, for example by evaluating $n!\mathsf{add}(n')$, a certain active object sends a request to integer object n to execute the method add with parameter n', then g_n^+, supplied with n' and the continuation f of the active object, is executed. We have that $g_n^+(n')(f)$ is, by definition, the parallel composition of f supplied with the immediate result of the execution of the method add, namely $n + n'$, and the process p_{St}, which remains unaltered: $g_n^+(n')(f) = f(n + n') \parallel p_{St}$. (A similar explanation applies to the presence in $p_{St}(\sigma)$ of the triples representing the Booleans.)

The standard objects are assumed to be present at the execution of every unit U. Therefore we adapt the denotational semantics of a unit (Definition 5.4) as follows:

$$[\![U]\!]_{\mathscr{D}} = \mathscr{D}_S[\![s_n]\!](\nu(\varnothing))(p_0) \parallel p_{St}.$$

III.3. Semantic equivalence. Finally, we extend the arguments presented in § 7 to show that for the modified versions of $[\![U]\!]_{\mathscr{O}}$ and $[\![U]\!]_{\mathscr{D}}$, as presented above, we still have

$$[\![U]\!]_{\mathscr{O}} = abstr([\![U]\!]_{\mathscr{D}}).$$

We begin by adapting the intermediate semantics $\mathscr{O}'_U$ (Definition 6.1), which will now be of type

$$\mathscr{O}'_U : \mathscr{P}_{fin}(LStat^*) \to \bar{P}_{cl}.$$

We put

$$\mathscr{O}'_U(\{S_t\}) = p_{St}$$

and for $X \subseteq LStat^* - \{S_t\}$ $(= LStat)$:

$$\mathcal{O}'_U(X \cup \{S_t\}) = \mathcal{O}'_U(X) \parallel \mathcal{O}'_U(\{S_t\}),$$

with $\mathcal{O}'_U(X)$ as defined according to Definition 6.1.

Next we extend the definition of *abstr* to an operation:

$$abstr^*: \bar{P}_{cl} \to (\Sigma \to \mathcal{P}(\Sigma_\partial^\infty)),$$

where *abstr** is defined as in Definition II.1. Please note, however, that for processes $p \in \bar{P}_{cl}$ it is in general *not* the case that $abstr^*(p)(\sigma)$ is a *closed* subset of Σ_∂^∞. Fortunately we can prove the following, which turns out to be all we need.

THEOREM III.1. *For every* $p \in \bar{P}_{co}$ *and* $\sigma \in \Sigma$: $abstr^*(p \parallel p_{St})(\sigma)$ *is compact.*

Proof. The proof is analogous to the one for Theorem II.3, given the additional observation that for every $p \in \bar{P}_{co}$ the set

$$(p \parallel p_{St})(\sigma) \cap (\Sigma \times \bar{P}_{cl})$$

is compact, which we prove now.

According to the definition of $\parallel$ we have

$$(p \parallel p_{St})(\sigma) = p(\sigma) \lfloor\!\lfloor p_{St} \cup p_{St}(\sigma) \lfloor\!\lfloor p \cup p(\sigma) \mid_\sigma p_{St}(\sigma).$$

From the continuity of $\parallel$ and the compactness of $p(\sigma)$ it follows that

$$(p(\sigma) \lfloor\!\lfloor p_{St}) \cap (\Sigma \times \bar{P}_{cl}) = \{\langle \sigma', p' \parallel p_{St} \rangle : \langle \sigma', p' \rangle \in p(\sigma)\}$$

is compact. Second, the set

$$(p_{St}(\sigma) \lfloor\!\lfloor p) \cap (\Sigma \times \bar{P}_{cl})$$

is empty. Finally, we show that

$$(p(\sigma) \mid_\sigma p_{St}(\sigma)) \cap (\Sigma \times \bar{P}_{cl})$$

is compact. Consider a sequence $(\langle \sigma, q_i \rangle)_i$ in this intersection. We show that it has a converging subsequence $(\langle \sigma, q_{k(i)} \rangle)_i$. According to the definition of $\mid_\sigma$ there exist sequences $(\langle \sigma_i, m_i, \beta_i, f_i, p_i \rangle)_i$ in $p(\sigma)$ and $(\langle \alpha_i, m_i, g_i \rangle)_i$ in $p_{St}(\sigma)$ such that

$$q_i = g_i(\beta_i)(f_i) \parallel p_i.$$

Because $p(\sigma)$ is compact there exists a monotonic function $k: \mathbb{N} \to \mathbb{N}$ such that

$$(\langle \alpha_{k(i)}, m_{k(i)}, \beta_{k(i)}, f_{k(i)}, p_{k(i)} \rangle)_i$$

is convergent. From the definition of the metric on $\bar{P}_{cl}$ it follows that we may assume that there exist α, m, and β such that for all i

$$\alpha_{k(i)} = \alpha, \quad m_{k(i)} = m, \quad \beta_{k(i)} = \beta.$$

The definition of p_{St} implies that for every $\langle \alpha, m, g \rangle$ in $p_{St}(\sigma)$ the function g is entirely determined by α and m. Thus

$$(\langle \alpha_{k(i)}, m_{k(i)}, g_{k(i)} \rangle)_i = (\langle \alpha, m, g_{k(i)} \rangle)_i = (\langle \alpha, m, g \rangle)_i,$$

for some g. Suppose we have

$$f = \lim_{i \to \infty} f_{k(i)} \wedge p = \lim_{i \to \infty} p_{k(i)};$$

then $\langle \alpha, m, \beta, f, p \rangle \in p(\sigma)$ and

$$\lim_{i \to \infty} \langle \sigma, q_i \rangle = \langle \sigma, g(\beta)(f) \parallel p \rangle \in (p(\sigma) \mid_\sigma p_{St}(\sigma)) \cap (\Sigma \times \bar{P}_{cl}). \qquad \square$$

COROLLARY III.2. $abstr^* \circ \mathcal{O}'_U \in \mathcal{P}_{fin}(LStat^*) \to P$.

(Recall that $P = \Sigma \to \mathcal{P}_{ncompact}(\Sigma_\partial^\infty)$.)

THEOREM III.3. $\mathcal{O}_U = abstr^* \circ \mathcal{O}'_U$.

This theorem can be proved by showing that in addition to $\mathcal{O}_U$ also $abstr^* \circ \mathcal{O}'_U$ is a fixed point of Φ_U. This can be done analogously to the proof of Theorem 7.2. From this observation and the fact that Φ_U is a contraction the theorem follows.

The definition of $\mathcal{D}^*_U$, which is given in Definition 7.5, is also changed. It will be a function of type

$$\mathcal{D}^*_U : \mathcal{P}_{fin}(LStat^*) \to \bar{P}_{cl},$$

that is, like the original $\mathcal{D}^*_U$ but for the clause that

$$\mathcal{D}^*_U(\{S_t\}) = p_{St}.$$

A last step toward the goal of this third Appendix, which is to prove the semantic equivalence of the denotational and operational semantics with standard objects present, consists of the observation that Theorem 7.7, stating that

$$\Phi'_U(\mathcal{D}^*_U) = \mathcal{D}^*_U,$$

can be proved for the new version of $\mathcal{D}^*_U$ as well. The extended proof involves some new case analysis (within Case 2), concerning the communications with standard objects. This being the last Appendix, this step being the last step towards our goal, and the author being only human, we omit the details and state without proof:

THEOREM III.4 (Extended version of Theorem 7.7). $\Phi'_U(\mathcal{D}^*_U) = \mathcal{D}^*_U$.

COROLLARY III.5 (Extended version of Corollary 7.8). $\mathcal{O}' = \mathcal{D}^*_U$.

Finally we are ready to prove the extended version of the main theorem, Theorem 7.10, of our paper.

THEOREM III.6. $[\![U]\!]_{\mathcal{O}} = abstr^*([\![U]\!]_{\mathcal{D}})$.

Proof.

$$\begin{aligned}
[\![U]\!]_{\mathcal{O}} &= \mathcal{O}_U[\![\{(\nu(\varnothing), s_n), S_t\}]\!] \\
&= [\text{Theorem III.3}] \\
&\quad abstr^*(\mathcal{O}'_U(\{(\nu(\varnothing), s_n), S_t\})) \\
&= [\text{Corollary III.5}] \\
&\quad abstr^*(\mathcal{D}^*_U(\{(\nu(\varnothing), s_n), S_t\})) \\
&= abstr^*(\mathcal{D}_S[\![s_n]\!](\nu(\varnothing))(p_0) \parallel p_{St}) \\
&= abstr^*([\![U]\!]_{\mathcal{D}}).
\end{aligned}$$

Acknowledgments. We wish to thank Pierre America for his detailed and constructive comments on preliminary versions of this paper. Discussions with Jaco de Bakker are gratefully acknowledged, as well as the contributions of the Amsterdam Concurrency Group: Jaco de Bakker, Frank de Boer, Arie de Bruin, Joost Kok, John-Jules Meyer, and Erik de Vink.

REFERENCES

[Am85] P. AMERICA, *Definition of the programming language* POOL-T, ESPRIT project 415, Doc. No. 0091, Philips Research Laboratories, Eindhoven, the Netherlands, 1985.

[Am86] ———, *Rationale for the design of* POOL, ESPRIT project 415, Doc. No. 0053, Philips Research Laboratories, Eindhoven, The Netherlands, 1986.

[Am87] ———, POOL-T—*A parallel object-oriented language*, in Object-Oriented Concurrent Systems, A Yonezawa and M. Tokoro, eds., MIT Press, Cambridge, MA, 1987.

[AB88] P. AMERICA AND J. W. DE BAKKER, *Designing equivalent semantic models for process creation*, Theoret. Comput. Sci., 60 (1988), pp. 109–176.

[ABKR86(a)] P. AMERICA, J. W. DE BAKKER, J. N. KOK, AND J. J. M. M. RUTTEN, *Operational semantics of a parallel object-oriented language*, in Conference Record of the 13th Symposium on Principles of Programming Languages, St. Petersburg, Florida, 1986, pp. 194–208.

[ABKR86(b)] ———, *A denotational semantics of a parallel object-oriented language*, Tech. Report CS-R8626, Centre for Mathematics and Computer Science, Amsterdam, the Netherlands, 1986, Inform. and Comput., to appear.

[ANSI83] ANSI, *Reference manual for the Ada programming language*, ANSI/MIL-STD 1815 A, U.S. Department of Defense, Washington DC, 1983.

[AR88] P. AMERICA AND J. J. M. M. RUTTEN, *Solving reflexive domain equations in a category of complete metric spaces*, in Proc. Third Workshop on Mathematical Foundations of Programming Language Semantics, M. Main, A. Melton, M. Mislove, and D. Schmidt, eds., Lecture Notes in Computer Science 298, Springer-Verlag, Berlin, New York, 1988, pp. 254–288; J. Comput. System Sci., to appear.

[Br86] A. DE BRUIN, *Experiments with continuation semantics: jumps, backtracking, dynamic networks*, Ph.D. thesis, Free University of Amsterdam, Amsterdam, the Netherlands, 1986.

[BBKM84] J. W. DE BAKKER, J. A. BERGSTRA, J. W. KLOP, AND J.-J. CH. MEYER, *Linear time and branching time semantics for recursion with merge*, Theoret. Comput. Sci., 34 (1984), pp. 135–156.

[BKMOZ86] J. W. DE BAKKER, J. N. KOK, J.-J. CH. MEYER, E.-R. OLDEROG, AND J. I. ZUCKER, *Contrasting themes in the semantics of imperative concurrency*, in Current Trends in Concurrency, J. W. de Bakker, W. P. de Roever, and G. Rozenberg, eds., Lecture Notes in Computer Science 224, Springer-Verlag, Berlin, New York, 1986, pp. 51–121.

[BZ82] J. W. DE BAKKER AND J. I. ZUCKER, *Processes and the denotational semantics of concurrency*, Inform. and Control, 54 (1982), pp. 70–120.

[Cl81] W. D. CLINGER, *Foundations of actor semantics*, Ph.D. thesis, AI-R-633, Massachusetts Institute of Technology, Cambridge, MA, 1981.

[Du66] J. DUGUNDJI, *Topology*, Allyn and Bacon, Boston, 1966.

[En77] E. ENGELKING, *General topology*, Polish Scientific Publishers, 1977.

[Go79] M. J. C. GORDON, *The Denotational Description of Programming Languages*, Springer-Verlag, Berlin, New York, 1979.

[HP79] M. HENNESSY AND G. D. PLOTKIN, *Full abstraction for a simple parallel programming language*, in Proceedings 8th MFCS, J. Bečvář, ed., Lecture Notes in Computer Science 74, Springer-Verlag, 1979, pp. 108–120.

[KR88] J. N. KOK AND J. J. M. M. RUTTEN, *Contractions in comparing concurrency semantics*, in Proc. 15th ICALP, Tampere, Lecture Notes in Computer Science 317, Springer-Verlag, Berlin, New York, 1988, pp. 317–332.

[Mi51] E. MICHAEL, *Topologies on spaces of subsets*, Trans. Amer. Math. Soc., 71 (1951), pp. 152–182.

[Mil80] R. MILNER, *A Calculus of Communicating Systems*, Lecture Notes in Computer Science 92, Springer-Verlag, Berlin, New York, 1980.

[Od87] E. A. M. ODIJK, *The* DOOM *system and its applications: a survey of* ESPRIT 415 *subproject* A, in Parallel Architectures and Languages Europe, Volume I, J. W. de Bakker, A. J. Nijman, and P. C. Treleaven, eds., Lecture Notes in Computer Science 258, Springer-Verlag, Berlin, New York, 1987, pp. 461–479.

[P176] G. D. PLOTKIN, *A powerdomain construction*, SIAM J. Comput., 5 (1976), pp. 452–487.

[P181] ———, *A structural approach to operational semantics*, Report DAIMI FN-19, Department of Computer Science, Aarhus University, 1981.

[P183] ———, *An operational semantics for* CSP, in Formal Description of Programming Concepts II, D. Bjørner, ed., North-Holland, Amsterdam, the Netherlands, 1983, pp. 199–223.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 384–396, April 1990

011

ON THE DECOMPOSABILITY OF *NC* AND *AC* *

CHRISTOPHER B. WILSON†

Abstract. It is shown for rationals $a, b \geq 1$ that $NC_a^{NC_b} = NC_{a+b-1}$. As a consequence, if, for some $k \geq 1$ and $\epsilon > 0$, $NC_k = NC_{k+\epsilon}$, then $NC_k = NC$. A similar development can be applied to circuits with unbounded fan-in. It is seen that $AC_a^{AC_b} = AC_{a+b}$, $AC_a^{NC_b} = NC_{a+b}$, and $NC_a^{AC_b} = AC_{a+b-1}$. This shows for $a \geq 0$ and $b \geq 1$ that $AC_a^{AC_b} = NC_{a+1}^{AC_b}$ and $AC_a^{NC_b} = NC_{a+1}^{NC_b}$. An oracle A is constructed for which $\forall k$, $NC_k^A \subset AC_k^A$ and, in fact, $AC_k^A - NC_{k+\epsilon}^A \neq \emptyset$ for any $\epsilon < 1$. Similarly, there is an A so that $\forall k$ and $\epsilon < 1$, $NC_{k+1}^A - AC_{k+\epsilon}^A \neq \emptyset$, and hence $AC_k^A \subset NC_{k+1}^A$. Combining, this yields an A such that, for all k and $0 < \epsilon < 1$, the classes AC_k^A and $NC_{k+\epsilon}^A$ are incomparable.

Key words. parallel complexity classes, NC, AC, circuits, size, depth, relativization

AMS(MOS) subject classification. 68Q15

1. Introduction.

In recent years the class NC has been established as one of the preferred characterizations of those problems with very fast parallel algorithms using a reasonable amount of hardware. It is a remarkably robust class, being invariant when defined on quite different models of computation. The most common models are parallel random access machines (PRAMs) with shared memory and uniform Boolean circuit families (see [5],[7]). As we are concerned here with some detailed structural theorems about the NC hierarchy, we will use the more basic model: that of circuits. The structural issue in question is whether it is possible to decompose any level of NC into components involving lower levels. We answer this affirmatively.

NC_k is defined to be the class of languages that are accepted by uniform families of circuits with bounded fan-in whose *size* grows polynomially with the length n of the input and whose *depth* grows proportional to $\log^k n$. The classes of particular interest are NC_a supplied with an oracle from NC_b. First we see that this is contained in NC_{a+b-1} so long as $b \geq 1$. Then for $a \geq 1$ and $b \geq 0$, NC_{a+b-1} can be expressed as NC_a relative to NC_b.

This provides characterizations of NC similar to those available for the polynomial time hierarchy PH. The polynomial hierarchy was originally defined in terms of one complexity class relative to another, and in [10] a characterization of each level was provided in terms of polynomially bounded alternating quantifiers applied to a polynomial time computable predicate. Here NC is defined in terms of increasing depth, and we show that there is an equivalent characterization in terms of one level of NC relative to another.

Using the above development, we get as a simple corollary a result first proved in [4]: for integers $k \geq 1$ if $NC_k = NC_{k+1}$, then $NC_k = NC$. This provides a structural analogy with the polynomial hierarchy: if $\Sigma_k^P = \Sigma_{k+1}^P$, then $\Sigma_k^P = PH$. However, the NC hierarchy need not be discrete, since we can surely examine NC_k for k rational or real (here it is rational for reasons of constructibility). As a corollary of the main result, we are able to show for rationals $k \geq 1$ and $\epsilon \geq 0$ that $NC_k = NC_{k+\epsilon}$ implies $NC_k = NC$.

The class AC_k is the same as NC_k except that unbounded fan-in gates are allowed.

* Received by the editors November 8, 1988; accepted for publication (in revised form) July 21, 1989. This work was partially supported by National Science Foundation grant CCR-8810051.

† Department of Computer and Information Science, University of Oregon, Eugene, Oregon 97403.

In §5 these techniques above are applied to the AC_k classes. It is shown that $AC_a^{AC_b} = AC_{a+b}$, $AC_a^{NC_b} = NC_{a+b}$, and $NC_a^{AC_b} = AC_{a+b-1}$. This has several interesting consequences, one being that the AC hierarchy may collapse in the same way as the NC hierarchy: for rationals $k \geq 0$ and $\epsilon > 0$, $k + \epsilon \geq 1$, if $AC_k = AC_{k+\epsilon}$, then $AC_k = AC$. Another consequence is for rationals $a \geq 0$ and $b \geq 1$ that $AC_a^{AC_b} = NC_{a+1}^{AC_b}$ and $AC_a^{NC_b} = NC_{a+1}^{NC_b}$. That is, an NC_{k+1} reduction provides no more power than an AC_k reduction when acting on any level of NC or AC.

In [12] an oracle A was constructed such that $\forall k$, $NC_k^A \subset NC_{k+1}^A$, where "$\subset$" denotes proper containment. In §6 we take that development considerably further. The same oracle A shows that $\forall k$ and $\epsilon < 1$, $AC_k^A - NC_{k+\epsilon}^A \neq \emptyset$. So in particular, $NC_k^A \subset AC_k^A$. A different approach to oracle-based languages allows us to construct an A for which $\forall k$ and $\epsilon < 1$, $NC_{k+1}^A - AC_{k+\epsilon}^A \neq \emptyset$, and hence $AC_k^A \subset NC_{k+1}^A$. The first construction allows an unbounded fan-in circuit to query a long sequence of large strings. The second construction takes advantage of the fact that a bounded fan-in circuit is charged less for a short query than for a long one, which is not true in the unbounded case. The two constructions can be combined to produce an A such that for any k and $0 < \epsilon < 1$, AC_k^A and $NC_{k+\epsilon}^A$ are incomparable.

2. Definitions and notation. A *Boolean circuit* is an acyclic directed graph whose nodes are labeled with an operator. Nodes of indegree zero are labeled as either *input* or *constant* gates, and those of outdegree zero are *output* gates. Nodes of indegree one are *negation* or *identity* gates, whereas those of indegree two are *and* or *or* gates. No nodes in the graph will be allowed to have indegree greater than two. A node may be both an output gate and perform an operation. Since we are primarily interested in deciding set membership, the circuits of use to us have only a single output gate. The circuit *accepts* an input string if the length of the string in binary is the same as the number of input gates of the circuit, and the circuit outputs the value one when given the string on its input gates.

The *size* of a circuit is the number of nodes it contains, and its *depth* is the length of the longest directed path in the graph. Intuitively, the size can be thought of as measuring the use of the hardware resource, whereas the depth is a measure of parallel time. A *circuit family* $\{\alpha_n\}$ accepts a set L if, for all n, α_n has n input nodes and accepts only those strings in L of length n. The circuit family has size $s(n)$ and depth $d(n)$ if $size(\alpha_n) = O(s(n))$ and $depth(\alpha_n) = O(d(n))$.

Another requirement put on circuit families is that of uniformity. This can be described in several ways, as covered in [8].

DEFINITION 1. A circuit family $\{\alpha_n\}$ is $u(n)$-uniform if there is a deterministic Turing machine which on any input of length n outputs an encoding of α_n using $O(u(n))$ workspace.

A circuit family is U_B uniform if it is $depth(\alpha_n)$-uniform, and it is U_{BC} uniform if it is $\log(size(\alpha_n))$-uniform. Other more complicated but technically more appealing notions of uniformity involve determining the structure of the circuit's connections on an alternating Turing machine [8]. We will adopt U_{BC} uniformity in the definitions below, though the exact choice of uniformity does not seem to affect our results.

DEFINITION 2. $NC_k = U(\log n)\text{-}DEPTH(\log^k n)$.

DEFINITION 3. $NC = \bigcup_{k=0}^{\infty} NC_k$.

There is a slight deviation in notation here. NC_k is normally written as NC^k here we will leave room for an oracle as superscript. Also notice that the size of the circuits is not explicitly restricted to be polynomial. This, however, follows automatically from the uniformity condition. A Turing machine operating in $O(\log n)$

space will only run for polynomial time (if it is going to halt), so it will only be able to output descriptions of polynomial size circuits.

The notion of allowing an NC circuit access to an oracle has been addressed in [5],[12]. The circuit is allowed *oracle gates*, through which it can determine the membership of a string in the oracle set. An oracle gate that has k input bits (sufficient for determining membership in the oracle of any string of length k) is defined to have *size* k and *depth* $\lceil \log_2 k \rceil$. If X is a set, we denote NC_k relative to X by NC_k^X. If $\mathcal{C}$ is a class of sets, then $NC_k^{\mathcal{C}}$ is $\bigcup_{X \in \mathcal{C}} NC_k^X$. The uniformity condition will be unaffected by the presence of an oracle. The Turing machine that acts as a constructor of the circuit family will not have access to the oracle. This issue is discussed in greater detail in [12].

This is directly analogous to the notion of NC_1-reducibility defined in [5], where the reduction is from function to function.

DEFINITION 4. A is NC_1-reducible to B, $A \leq^{NC_1} B$, if and only if $A \in NC_1^B$.

An unbounded fan-in circuit, which we will abbreviate hereafter as a UBF circuit, is a Boolean circuit as above but with no restriction on the indegree of any node. A class directly analogous to NC can be defined on this model.

DEFINITION 5. $AC_k = U(\log n)$–UBF $DEPTH(\log^k n)$.

DEFINITION 6. $AC = \bigcup_{k=0}^{\infty} AC_k$.

It is not too hard to see that for any $k \geq 0$, $NC_k \subseteq AC_k \subseteq NC_{k+1}$. A discussion of this class and related issues can be found in [5]. It is also known that the $PARITY$ function is not in AC_0 ([6]). This separates AC_0 from NC_1 and AC_1, but it is the only known such separation, aside from the obvious $NC_0 \neq AC_0$.

To handle an oracle in AC_k, we will allow oracle gates as above. As in [3], however, the size and depth of this gate is 1. (This is similar also to [4].) Intuitively, this adheres to the spirit of unbounded fan-in, as we can also compute the *and* of k bits in depth 1. On a bounded fan-in model, the k-bit *and* requires depth $\log k$, as it does to determine membership of a k-bit string in an oracle.

All logarithms are to the base two. In fact, by $\log n$ we will mean the function $\max(1, \lceil \log_2 n \rceil)$.

3. Oracles from *NC*. Initially we address the issue of allowing NC to have oracles from NC. That is, we want to know how complex a set in $NC_a^{NC_b}$ can be. This has been partially addressed in [5], where it is seen that NC_k is closed under $\leq^{NC_1}$, which is to say, $NC_1^{NC_k} = NC_k$. Adapting the proof of that result yields $NC_a^{NC_b} \subseteq NC_{ab}$. This can be considerably sharpened.

THEOREM 3.1. *For* $b \geq 1$, $NC_a^{NC_b} \subseteq NC_{a+b-1}$.

Proof. Consider an $O(\log^a n)$ depth circuit with queries made to a language in NC_b. Look at any path from an input to the output. It has length $O(\log^a n)$, so the series of queries $q_1, \cdots, q_k$ made on that path must satisfy $\sum_{i=1}^{k} \log |q_i| \leq c \log^a n$. Each query q_i has length at most polynomial in n, so $\log |q_i| \leq d \log n$. We will replace each query with an NC_b circuit. In the new circuit, after replacement, the length of the path will be the maximum of $O(\log^a n)$ (from the part of the path excluding the queries) and $\sum_{i=1}^{k} e \log^b |q_i|$ (caused by replacing the query by an NC_b circuit). The latter value can be bounded.

$$\sum_{i=1}^{k} e \log^b |q_i| \quad = \quad e \sum_{i=1}^{k} \log^{b-1} |q_i| \cdot \log |q_i|$$

$$\begin{aligned} &\leq e\sum_{i=1}^{k}(d\log n)^{b-1}\log|q_i| \\ &= e(d\log n)^{b-1}\sum_{i=1}^{k}\log|q_1| \\ &\leq e(d\log n)^{b-1}c\log^a n \\ &= O((\log n)^{a+b-1}). \end{aligned}$$

Since $b \geq 1$, we have $a + b - 1 \geq a$, so the depth of the entire path becomes $O((\log n)^{a+b-1})$. □

To show a containment in the other direction, we must take a uniform circuit family and be able to divide each circuit into arbitrarily sized uniform subcircuits.

THEOREM 3.2. *$NC_{a+b-1} \subseteq NC_a^{NC_b}$ for rationals $a \geq 1$ and $b \geq 0$.*

Proof. Let us assume that $b \geq 1$. Otherwise, if $0 \leq b < 1$, then trivially $NC_{a+b-1} \subseteq NC_a \subseteq NC_a^{NC_b}$.

Given a language $S \in NC_{a+b-1}$, look at the family $\{\alpha_n\}$ of circuits accepting it. The α_n are generable by a Turing machine M_α in space $O(\log n)$ from any input of length n, and each α_n has depth $c(\log n)^{a+b-1}$ for some constant c. Suppose that M_α on an input of size n outputs a sequence of p tuples of the form (i, l_i, r_i, t_i), where l_i and r_i are the inputs to gate i and t_i indicates the type of gate i. In a fixed α_n, let *dist*(i) be the maximum distance of gate i from an input gate.

We will demonstrate the existence of an NC_a circuit family using an oracle from NC_b and behaving the same as $\{\alpha_n\}$. This will show that $S \in NC_a^{NC_b}$. The idea is to divide each α_n into $c\log^{a-1} n$ levels, each of depth $\log^b n$. We then provide the computation of each level as an oracle. Since each query will have at most polynomial length, the depth of each query will be $O(\log n)$. There are at most $c\log^{a-1} n$ of them in any series, so the resulting relativized circuit has depth $O(\log^a n)$. This NC_a circuit family is easily seen to be $\log n$-uniform.

A problem arises with the NC_b subcircuits. It is not at all clear that they are uniform, since generating them requires determining the depth of each node in α_n. This seems to require transitive closure, a problem complete for NL and not known to be solvable in $O(\log n)$ deterministic space. Instead we use a technique found in Theorem 12 of [1] to construct from α_n an equivalent circuit β_n with a very regular structure that can be easily subdivided.

Let $K = c(\log n)^{a+b-1}$ be an upper bound to the depth of α_n. Also let $p = p(n)$ bound the number of gates in α_n. The circuit β_n consists of K levels, each of p gates, one for each gate of α_n. Each level takes all its inputs from the previous level. Each level L_j consists of gates $g_{j,1}, g_{j,2}, \cdots, g_{j,p}$.

In L_0, if gate i is an input node, then gate $g_{0,i}$ will be a single gate, labeled as an input gate. Otherwise, gate $g_{0,i}$ will be a constant zero gate.

At level L_j, for $j \geq 1$, each gate $g_{j,i}$ will take as inputs g_{j-1,l_i} and g_{j-1,r_i}. That gate will compute $g_{j-1,l_i} \diamond g_{j-1,r_i}$, where $\diamond$ is the operation determined by t_i. Other gates of indegree one and zero can be handled in an entirely similar manner. However, if i is an input gate in α_n, then $g_{j,i}$ will perform the *identity* operation on input $g_{j-1,i}$. This way, the original input values are passed from level to level.

Figure 1(a) illustrates a sample circuit. Figure 1(b) shows the transformation of that circuit. The circled nodes indicate which nodes are relevant in the simulation of the original circuit.

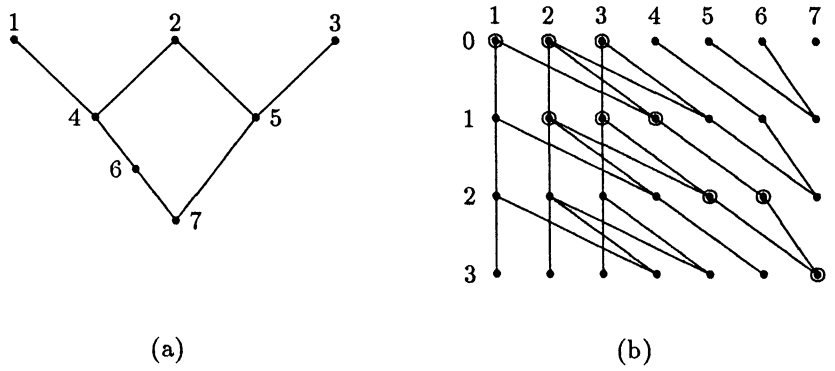


FIG. 1. *A circuit and its transformation.*

The fact that β_n performs the same as α_n can be shown by induction on the depth of the gates of α_n. This follows from

Claim. If $dist(i) \leq j$, then gate $g_{j,i}$ of β_n outputs the same value as gate i of α_n.

The claim is seen by induction on j. If $dist(i) = 0$, then i is an input and gate $g_{0,i}$ provides that input. Suppose then that $dist(i) \leq j$ for $j \geq 1$. If i is an input gate in α_n, then by definition $g_{j,i}$ provides that value. For i not an input, look at its left l_i and right r_i predecessors. Since $dist(l_i) \leq j - 1$ and $dist(r_i) \leq j - 1$, by the inductive hypothesis g_{j-1,l_i} and g_{j-1,r_i} yield the same values as l_i and r_i in α_n. Gate $g_{j,i}$ computes the same operation on those values as gate i in α_n does.

The circuit family $\{\beta_n\}$ thus accepts S. Each β_n is split into subcircuits B_0, $B_1, \cdots, B_{c \log^{a-1} n}$. B_0 is simply L_0. Subcircuit B_l consists of levels $L_{(l-1)\log^b n+1}$ through $L_{l \log^b n}$. Each of these B_l is now $\log n$-uniform, so each computes a function f_l in NC_b. Here we see the need for a and b to be rational, for otherwise those levels could not necessarily be determined. The language

$$F = \{ < y, i, l > \mid \text{the } i\text{th bit of } f_l(y) \text{ is } 1 \}$$

is also in NC_b, since $b \geq 1$. Using F as an oracle, there will be an NC_a circuit accepting S. □

In the previous proof, notice that we split up the circuit and provide each level as an NC_b oracle. Unfortunately, each of these correspond to separate languages, but the circuit can query only one language. As in [4], we provide information about all the languages encoded into the language F. To perform the selection, we need at least $O(\log n)$ depth, which is why $b \geq 1$ was necessary in the construction above. This problem also arises in Theorem 5.2(b)-(c) when dealing with AC.

The method of encoding the problem of computing a function as a set recognition problem used above could also be applied more generally. If NC and AC were viewed as classes of *functions*, as they normally are, then the structural decompostions of these classes derived in this paper could still be shown to hold.

4. Main theorem and applications. The main theorem follows directly from Theorems 3.1 and 3.2.

THEOREM 4.1. *For rationals $a \geq 1$ and $b \geq 1$, $NC_{a+b-1} = NC_a^{NC_b}$.*

As corollaries to Theorem 4.1, we get fairly simple proofs of some other structural properties.

COROLLARY 4.2. [5]. *NC_k is closed under $\leq^{NC_1}$ for $k \geq 1$.*

Proof. $NC_1^{NC_k} = NC_{1+k-1} = NC_k$ □

COROLLARY 4.3. *Let $k \geq 1$ be rational. If $NC_k = NC_{k+1}$, then $NC_k = NC$.*

Proof. First note that $NC_{k+2} = NC_2^{NC_{k+1}} = NC_2^{NC_k} = NC_{k+1} = NC_k$. The fact that $NC_k = NC$ follows by induction via this process. □

Corollary 4.3 has been proved in [4], holding only when k is an integer. Our characterizations provide for a somewhat more general case. We can also show under weakened assumptions that NC will still collapse. Consider, for example, the assumption that $NC_{3.5} = NC_4$. Then

$$NC_{4.5} = NC_{1.5}^{NC_4} = NC_{1.5}^{NC_{3.5}} = NC_4 = NC_{3.5},$$

so by Corollary 4.3, $NC_{3.5} = NC$. This can be generalized.

THEOREM 4.4. *Let $k \geq 1$ and $\epsilon > 0$ be rationals. If $NC_k = NC_{k+\epsilon}$, then $NC_k = NC$.*

Proof. We will start to decompose NC_{k+1}:

$$\begin{aligned} NC_{k+1} &= NC_{2-\epsilon}^{NC_{k+\epsilon}} = NC_{2-\epsilon}^{NC_k} = NC_{k+1-\epsilon} \\ &= NC_{2-2\epsilon}^{NC_{k+\epsilon}} = NC_{2-2\epsilon}^{NC_k} = NC_{k+1-2\epsilon} \\ &= \cdots = NC_{2-c\epsilon}^{NC_k} = NC_{k+1-c\epsilon} \end{aligned}$$

after c iterations of this process. We must choose an integer c so that $2 - c\epsilon \geq 1$ (recall that Theorem 4.1 needs $a \geq 1$) and $k + 1 - c\epsilon \leq k + \epsilon$. Choosing c in the range $\epsilon^{-1} - 1 \leq c \leq \epsilon^{-1}$ will work. Therefore

$$NC_{k+1} = NC_{k+1-c\epsilon} \subseteq NC_{k+\epsilon} = NC_k.$$

Corollary 4.3 now yields the fact that $NC_k = NC$. □

The powers need not always be rational. The theorems above would hold for reals satisfying certain space-constructibility constraints (such as requiring a sufficient number of bits, dependent on the length of the input, to be computable in $O(\log n)$ space). The previous theorem also holds for arbitrary reals.

COROLLARY 4.5. *Let $\alpha \geq 1$ and $\delta > 0$ be real numbers. If $NC_\alpha = NC_{\alpha+\delta}$, then $NC_\alpha = NC$.*

Proof. If $\alpha = 1$, then it is obviously rational. Pick rational $\epsilon \in (0, \delta]$ and apply Theorem 4.4. If $\alpha > 1$, then choose rationals k and ϵ satisfying $k \geq \alpha$, $\epsilon > 0$, and $k + \epsilon \leq \alpha + \delta$. This can be achieved by picking $k \in [\alpha, \alpha + (\delta/2)]$ and $\epsilon \in (0, \delta/2]$. Note that $k + \epsilon \leq \alpha + (\delta/2) + (\delta/2) = \alpha + \delta$.

Then

$$NC_{k+\epsilon} \subseteq NC_{\alpha+\delta} = NC_\alpha \subseteq NC_k.$$

Because $NC_k = NC_{k+\epsilon}$ and $k \geq 1$, we can apply Theorem 4.4 and see that

$$NC = NC_k = NC_{k+\epsilon} \subseteq NC_{\alpha+\delta} = NC_\alpha.$$ □

Theorem 4.4 can, under a different assumption, show a collapse of NC below NC_1.

COROLLARY 4.6. *Let $k \geq 0$ and $\epsilon > 0$ be rationals, $k + \epsilon > 1$. If $NC_k = NC_{k+\epsilon}$, then $NC_k = NC$.*

Proof. If $k \geq 1$, then Theorem 4.4 applies. If $k < 1$, then $NC_{k+\epsilon} = NC_k \subseteq NC_1$. In other words, $NC_1 = NC_{1+\xi}$, where $\xi = k + \epsilon - 1 > 0$. Theorem 4.4 shows that $NC = NC_1 \subseteq NC_{k+\epsilon} = NC_k$. □

5. Unbounded fan-in circuits. In this section we apply the previous development to the class AC. First, we will point out that the standard containment of AC_k in NC_{k+1} holds for any oracle.

LEMMA 5.1. *For any oracle A and $k \geq 0$, $NC_k^A \subseteq AC_k^A \subseteq NC_{k+1}^A$.*

Proof. The first containment is trivial. To see that $AC_k^A \subseteq NC_{k+1}^A$, we can expand any *and* and *or* gate into a bounded fan-in tree. Charging $\log s$ rather than 1 for the depth of an oracle gate of size s will increase the depth by a factor of $O(\log n)$, since s is at most a polynomial in n. For a careful handling of the uniformity issue, the reader is referred to [8]. □

Similar to Theorems 3.1 and 3.2, we can see what happens when we give NC_b and AC_b to NC_a and AC_a as oracles.

THEOREM 5.2.

(a) $AC_a^{AC_b} \subseteq AC_{a+b}$ *for* $a \geq 0$ *and* $b \geq 0$.

(b) $AC_{a+b} \subseteq AC_a^{AC_b}$ *for rationals* $a \geq 0$ *and* $b \geq 1$.

(c) $NC_{a+b} \subseteq AC_a^{NC_b}$ *for rationals* $a \geq 0$ *and* $b \geq 1$.

(d) $NC_a^{AC_b} \subseteq AC_{a+b-1}$ *for* $a \geq 1$ *and* $b \geq 1$.

Proof. (a) In a UBF circuit of depth $O(\log^a n)$, look at the series of queries to AC_b made on a path from an input to the output. There are at most $O(\log^a n)$ of them, since each has depth one. At worst, they are polynomial in size, so we will replace each by a UBF circuit of depth $O(\log^b n)$. The resulting UBF circuit is $\log n$-uniform and has depth $O((\log n)^{a+b})$.

(b) The uniform leveling technique in Theorem 3.2 applies just as well to UBF circuits. In the same way, we can take a circuit of depth $O((\log n)^{a+b})$ and split it into $O(\log^a n)$ levels (each $\log n$-uniform) of depth $\log^b n$. Each computes a function f_l $(1 \leq l \leq O(\log^a n))$ in AC_b. The set

$$F = \{ \langle y, i, l \rangle \mid \text{the } i \text{ th bit of } f_l(y) \text{ is } 1 \}$$

is also in AC_b so long as $b \geq 1$. The queries to F have depth one, so the set accepted by the original circuit family is in AC_a^F.

(c) The same leveling technique used in Theorem 3.2 and in the previous paragraph applies here. The $O(\log^a n)$ queries to NC_b are of polynomial size but are made by a UBF circuit in this context. Each will have depth one.

(d) Here we replace a series of queries in the NC_a circuit by AC_b circuits. The result will be a UBF circuit. Its depth will be $O((\log n)^{a+b-1})$ by an appeal to the techniques of Theorem 3.1. □

Now we are able to provide other characterizations of NC_k and AC_k as in Theorem 4.1.

THEOREM 5.3. *Let a and b be rationals.*

(a) $AC_a^{AC_b} = AC_{a+b}$ *for* $a \geq 0$ *and* $b \geq 1$.

(b) $AC_a^{NC_b} = NC_{a+b}$ *for* $a \geq 0$ *and* $b \geq 1$.

(c) $NC_a^{AC_b} = AC_{a+b-1}$ *for* $a \geq 1$ *and* $b \geq 1$.

Proof. Part (a) follows from Theorem 5.2(a)-(b). Part (b) in one direction follows from Theorem 5.2(c). To get containment in the other direction, we notice that

$$AC_a^{NC_b} \subseteq NC_{a+1}^{NC_b} \subseteq NC_{a+b}$$

by Lemma 5.1 and Theorem 3.1. Part (c) is obtained by use of Theorem 5.2(d) and the fact that

$$AC_{a+b-1} \subseteq AC_{a-1}^{AC_b} \subseteq NC_a^{AC_b}$$

by Theorem 5.2(b) and Lemma 5.1. □

The AC hierarchy collapses like the NC hierarchy. Modifying the proof of Theorem 4.4 suffices to show this.

COROLLARY 5.4. *Let $k \geq 0$ and $\epsilon > 0$ be rationals, $k + \epsilon \geq 1$. If $AC_k = AC_{k+\epsilon}$, then $AC_k = AC$.*

An interesting point that Theorem 5.3 shows is that AC_a is equal to NC_{a+1} if oracles from AC_b or NC_b are used.

COROLLARY 5.5. *Let $a \geq 0$ and $b \geq 1$ be rationals.*

(a) $AC_a^{AC_b} = NC_{a+1}^{AC_b}$

(b) $AC_a^{NC_b} = NC_{a+1}^{NC_b}$

Proof. By Theorems 4.1 and 5.3, $AC_a^{AC_b} = AC_{a+b} = NC_{a+1}^{AC_b}$ and $AC_a^{NC_b} = NC_{a+b} = NC_{a+1}^{NC_b}$. □

We see that if oracles are chosen from AC_b or NC_b for $b \geq 1$, then AC_a offers no less power than NC_{a+1}. This seems counterintuitive in light of the fact that AC_0 is properly contained in NC_1. We also see that $AC_k^{NC_1} = NC_{k+1}^{NC_1} = NC_{k+1}$. Similarly, $NC_k^{AC_1} = AC_k$. Note that this does not say that any NC_{a+1} reduction can be replaced with an AC_a reduction on sets from AC_b or NC_b. It does say that if A NC_{a+1}-reduces to $B \in NC_b$, then there is a $C \in NC_b$ such that A AC_a-reduces to C. An interesting question is the relationship between B and C.

At lower levels of the NC and AC hierarchies, we are especially interested in the containment $NC_1 \subseteq AC_1 \subseteq NC_2$. Theorem 5.3 shows that $AC_1 = NC_1^{AC_1}$ and $NC_2 = AC_1^{NC_1}$. We could view this as evidence that AC_1 is "closer to" NC_2 than NC_1, since it needs a weaker oracle. However, AC_1 has more power in accessing the oracle than does NC_1, so we must be careful in making such interpretations.

In [4] there is a characterization of NC as a hierarchy similar to the polynomial hierarchy:

$$\Sigma_1^{NC} = NC_1 \text{ and } \Sigma_{k+1}^{NC} = AC_1^{\Sigma_k^{NC}}.$$

It is shown that for all integers $k \geq 1$ that $\Sigma_k^{NC} = NC_k$. This follows from Theorem 5.3(b) as well. Also, AC_1 can be replaced by NC_2. Similarly, each AC_k could be defined in this manner:

$$\Sigma_1^{AC} = AC_1 \text{ and } \Sigma_{k+1}^{AC} = AC_1^{\Sigma_k^{AC}} (= NC_2^{\Sigma_k^{AC}}).$$

We have seen that if $NC_k = NC_j$ ($AC_k = AC_j$) for $k < j$, then NC (AC) collapses. A natural question is what happens if either $NC_k = AC_k$ or $AC_k = NC_{k+1}$. We cannot exhibit a hierarchy collapse, but the equalities translate upwards.

COROLLARY 5.6. *Let $k \geq 1$ be rational.*

(a) *If $NC_k = AC_k$, then, for all rational $j \geq k$, $NC_j = AC_j$.*

(b) *If $AC_k = NC_{k+1}$, then, for all rational $j \geq k$, $AC_j = NC_{j+1}$.*

Proof. For part (a), assume that $NC_k = AC_k$. For any rational $\delta \geq 0$,

$$AC_{k+\delta} = AC_\delta^{AC_k} = AC_\delta^{NC_k} = NC_{k+\delta}.$$

Similarly for part (b), assume that $AC_k = NC_{k+1}$. For any rational $\delta \geq 0$,

$$NC_{k+1+\delta} = NC_{1+\delta}^{NC_{k+1}} = NC_{1+\delta}^{AC_k} = AC_{k+\delta}.$$

These follow directly from Theorems 4.1 and 5.3. □

6. Separation with oracles. In [12] there is exhibited an oracle A so that, for all k, $NC_k^A \neq NC_{k+1}^A$. Here we will apply the same method to separate NC_k from AC_k and AC_k from NC_{k+1}.

A language $L_{k+1}(A)$ was introduced in [12] having the property that $\forall A$, $L_{k+1}(A) \in NC_{k+1}^A$. A specific oracle A was then constructed so that $\forall k$, $L_{k+1}(A) \notin NC_k^A$. It turns out for any k and oracle A that $L_{k+1}(A) \in AC_k^A$. This suffices to give a relativized separation of NC_k and AC_k.

THEOREM 6.1. *There exists a recursive oracle A so that for any rational $k \geq 0$ and $0 \leq \epsilon < 1$, $AC_k^A - NC_{k+\epsilon}^A \neq \emptyset$.*

Proof. We will describe the language $L_{k+1}(A)$ and then point out why it is in AC_k^A. $L_{k+1}(A)$ is the set accepted by the following procedure.

Input x, $|x| = n$:
  $K \leftarrow \lceil \log^{k+1} n \rceil$
  for $i \leftarrow 1$ to $K - 1$ do
    if $x0^{n-i}1b_{i-1}\cdots b_1 \in A$ then $b_i \leftarrow 1$
      else $b_i \leftarrow 0$
  if $x0^{n-K}1b_{K-1}\cdots b_1 \in A$
    then **accept** x
    else **reject** x.

A cursory examination of this algorithm would seem to indicate that $L_{k+1}(A)$ is in AC_{k+1}^A. We can do better if we are careful. A UBF circuit can determine $\log n$ bits b_i at a time, so the maximum depth need only be $O(\log^k n)$. To see that $\log n$ bits can be determined in constant depth and polynomial size, let us illustrate how to find the first $\log n$ bits b_i. Let binary strings γ have length $\log n$: $\gamma = \gamma_{\log n}\cdots\gamma_1$. Define $f_\gamma = \bigwedge_{i=1}^{\log n}(x0^{n-i}1\gamma_{i-1}\cdots\gamma_1 \in A)^{\gamma_i}$, where $p^1 = p$ and $p^0 = \neg p$ for Boolean variable p. Then $b_i = \bigvee_{\gamma,\gamma_i=1} f_\gamma$. A similar process can be repeated to find the next $\log n$ bits in constant depth and polynomial size, and so on. Thus, for all k and A, $L_{k+1}(A) \in AC_k^A$.

Let $\langle\cdot,\cdot\rangle$ be a standard pairwise encoding function on integers. This can be extended to handle more integers by composition. The ith circuit family is the one constructed by machine M_i, where $M_1, M_2, \cdots$ is an enumeration of $O(\log n)$ space transducers. At stage $e = \langle i, c, p, q, s, t\rangle$, letting $k = p/q$ and $\epsilon = s/t$ we ensure that if M_i constructs a circuit family of depth at most $c\log^{k+\epsilon} n$, then that family cannot accept $L_{k+1}(A)$. This is done by choosing n appropriately, ensuring on an input of length n that M_i constructs an α of depth at most $c\log^{k+\epsilon} n$, and, if so, diagonalizing across the behavior of α. Initially, A will be empty, and strings will be added to it. Once added, no string will ever be removed.

For the moment we are only concerned about queries of length $2n$, those that are relevant to membership in $L_{k+1}(A)$. Given a circuit α, we will break it up into *independent query levels.* Level 1 consists of those queries that depend on no other query (that is, no other query lies on a directed path ending at that query). Level j consists of those queries that depend on some query from level $j-1$. If α has depth $c\log^{k+\epsilon} n$, then it can have at most $(c\log^{k+\epsilon} n)/(\log 2n) \leq c\log^{k+\epsilon-1} n$ such levels.

Construction of A. Stage $e = \langle i, c, p, q, s, t\rangle$.

Check that $q \neq 0$, $t \neq 0$, and $\epsilon = s/t < 1$. If not, skip this stage. Let $k = p/q$. Choose n large enough to satisfy the following constraints:

- $2^{(\log^{2-\epsilon} n)/c}$ is larger than the polynomial that bounds the size of the circuit constructed by M_i on 0^n and
- $2n$ is larger than anything queried or added to A at any previous stage.

Let α be the circuit constructed by M_i on 0^n. If the depth of α exceeds $c\log^{k+\epsilon} n$, then skip this stage. We now proceed in steps, and at each step fix

$$(\log^{k+1} n)/(c\log^{k+\epsilon-1} n) = (\log^{2-\epsilon} n)/c$$

bits b_i. The steps will be numbered 1 through $c\log^{k+\epsilon-1} n$. Fix $x = 0^n$ as the input to α.

Step m. Let $j = ((m-1)/c)\log^{2-\epsilon} n$. (Invariant: No string of the form $xzb_j \cdots b_1$, $|z| = n - j$, has been queried by α at levels 1 through $m-1$.) There are $2^{(\log^{2-\epsilon} n)/c}$ strings y of length $(\log^{2-\epsilon} n)/c$. By the first constraint in the choice of n there must be some y for which no string of the form $xzyb_j \cdots b_1$, $|z| = n - |y| - j$, has been queried at this or any previous level. Pick such a y and for $l = 1$ to $(\log^{2-\epsilon} n)/c$ put $x0^{n-l-j}1y_{l-1}\cdots y_1 b_j \cdots b_1$ into A if and only if y_l=1. (*End step m.*)

Now that we have dealt with all levels of α, we can add strings of the form $xzb_{K-1}\cdots b_1$, $K = \log^{k+1} n$ and $|z| = n - K + 1$, to A without affecting the behavior of α on x. The final step is to add $x0^{n-K}1b_{K-1}\cdots b_1$ to A if and only if α with oracle A rejects x. **End construction.**

Adding the final string to A cannot affect the behavior of α on x due to the invariance condition.

The first constraint in the choice of n provides further assurance that α will be unable to accept $L_{k+1}(A)$. Since it is certainly true that $2^{(\log^{2-\epsilon} n)/d} > n$, we must have $(\log^{2-\epsilon} n)/d > \log n$ or $d < \log^{1-\epsilon} n$. This implies that $d\log^{k+\epsilon} n < \log^{k+1} n$, the latter value being the depth within which an NC circuit could simulate the AC_k circuit accepting $L_{k+1}(A)$ described above. □

COROLLARY 6.2. *There exists an oracle A so that for any rational $k \geq 0$, $NC_k^A \subset AC_k^A$, where "$\subset$" denotes proper containment.*

An oracle was able to witness a separation between NC_k and AC_k, since an AC_k circuit is able to ask a dependent series of $O(\log^k n)$ questions, each of length $O(n)$. An NC_k circuit is not always able to do this. If we want to separate AC_k and NC_{k+1}, we will have to look at some advantage NC_{k+1} has over AC_k. One advantage is that it can ask a series of $O(\log^{k+1} n/\log\log n)$ questions, each of length $O(\log^2 n)$. A bounded fan-in circuit benefits from asking shorter questions, whereas an unbounded fan-in circuit has no easy way to do this.

THEOREM 6.3. *There exists a recursive oracle A so that, for any rational $k \geq 0$ and $0 \leq \epsilon < 1$, $NC_{k+1}^A - AC_{k+\epsilon}^A \neq \emptyset$.*

Proof. Consider the language $S_{k+1}(A)$ described by the following algorithm:

Input x, $|x| = n$:
  $x_0 \leftarrow 0^{\log^2 n}$
  $K \leftarrow \log^2 n$, $L \leftarrow \lceil \frac{\log^{k+1} n}{\log\log n} \rceil$
  for $i \leftarrow 1$ to L do begin
    for every $1 \leq j \leq K$ do *in parallel*

if $x_{i-1}0^{K-j}10^{j-1} \in A$ then $b_j \leftarrow 1$
else $b_j \leftarrow 0$
end *parallel*
$x_i \leftarrow b_K b_{K-1} \cdots b_1$
end
if $x_L 0^K \in A$
then **accept** x
else **reject** x.

On an NC circuit, the depth to determine membership in $S_{k+1}(A)$ is

$$\left(\frac{\log^{k+1} n}{\log\log n} + 1\right) \cdot \log(2\log^2 n) = O(\log^{k+1} n).$$

So for any A, $S_{k+1}(A) \in NC^A_{k+1}$. The obvious UBF circuit for $S_{k+1}(A)$ has depth $(\log^{k+1} n)/(\log\log n)$. We will show how to construct an oracle A such that, for all k, $S_{k+1}(A) \notin AC^A_{k+\epsilon}$.

Similar to the previous construction, at stage $e = \langle i, c, p, q, s, t\rangle$ if M_i constructs a circuit family of depth no more than $c\log^{k+\epsilon} n$, where $k = p/q$ and $\epsilon = s/t$, then we will ensure that this family will not accept $S_{k+1}(A)$. Initially, A will be empty, and strings will be added to it. Once added, no string will ever be removed.

Given a circuit α, we will break it up into independent query levels, as above. The number of independent query levels in a UBF circuit is clearly bounded above by its depth.

Construction of A. Stage $e = \langle i, c, p, q, s, t\rangle$.

Check that $q \neq 0$, $t \neq 0$, and $\epsilon = s/t < 1$. If not, skip this stage. Let $k = p/q$.

Choose n large enough to satisfy the following constraints:

- $\frac{\log^{k+1} n}{\log\log n} > c\log^{k+\epsilon} n$.
- $2^{\log^2 n}$ exceeds $c\log^{k+\epsilon} n$ plus the size (a polynomial) of the circuit constructed by M_i on 0^n.
- $2\log^2 n$ is larger than anything queried or added to A at any previous stage.

Let α be the UBF circuit constructed by M_i on 0^n. If the depth of α is larger than $c\log^{k+\epsilon} n$, then skip the rest of this stage. Fix 0^n as the input to α. This stage proceeds in Steps 1 through $(\log^{k+1} n)/(\log\log n)$. Initially, let $x_0 = 0^{\log^2 n}$, $K = \log^2 n$, and $L = (\log^{k+1} n)/(\log\log n)$.

Step i. Find an x_i of length $\log^2 n$ and $\forall j < i$, $x_i \neq x_j$ so that for no z of length $\log^2 n$ is $x_i z$ queried at levels 1 through i of α. This must exist, since $2^{\log^2 n}$ is larger than the size of the circuit plus the number of its query levels. Where $x_i = b_K \cdots b_1$, add $x_{i-1}0^{K-j}10^{j-1}$ to A for each j satisfying $b_j = 1$. (*End step i.*)

Finally, if α rejects 0^n, then add $x_L 0^K$ to A. If α accepts, do not add it to A. **End construction.**

Note that L is larger than the number of independent query levels of α, so by construction α cannot have queried $x_L 0^K$. For the A described by the construction, it is the case for every rational k and $\epsilon < 1$ that any $AC^A_{k+\epsilon}$ circuit family cannot accept $S_{k+1}(A)$. □

COROLLARY 6.4. *There exists an oracle A so that, for any rational $k \geq 0$, $AC_k^A \subset NC_{k+1}^A$, where "$\subset$" denotes proper containment.*

In fact, we have shown that $NC_{k+1}^A - AC_{k+\epsilon}^A$ contains a tally set. Another fact worth noting is that the proofs of the two previous theorems could be interleaved to construct an A where for all k, $NC_k^A \subset AC_k^A \subset NC_{k+1}^A$. As an even stronger result, we can get the following.

COROLLARY 6.5. *There exists an oracle A such that for all rationals $k \geq 0$ and $0 < \epsilon < 1$, AC_k^A and $NC_{k+\epsilon}^A$ are incomparable.*

In [2],[13] there is introduced a notion of relativized space that is a reasonable measure to compare with relativized depth. One is referred to the original papers for details, but essentially the oracle Turing machine can put partially constructed queries into some storage mechanism, say a stack. In this way we define, for an oracle A, the classes sL^A, stack log-space relative to A, and sNL^A, the nondeterministic version (an important consideration here is that the nondeterministic machine must act deterministically while the stack is not empty). In [13] it is shown, for any A, that $NC_1^A \subseteq sL^A$ and $sNL^A \subseteq NC_3^A$. The proof of the latter containment can easily be modified to show that $sNL^A \subseteq AC_2^A$. Compare these to the unrelativized $NC_1 \subseteq L \subseteq NL \subseteq AC_1 \subseteq NC_2$.

Corollary 6.5 now shows that there is an oracle A so that, for any $\epsilon < 1$, $sL^A - AC_\epsilon^A$ is not empty, because NC_1^A and AC_ϵ^A are incomparable. This indicates that it may be difficult if not unlikely to improve upon the containment $NL \subseteq AC_1$. By improvement, we mean in terms of depth, as it is known that NL is contained in the semi-unbounded class SAC_1 [11]. Similarly, for any $\epsilon < 1$, $NC_{2+\epsilon}^A - sNL^A$ is not empty.

7. Conclusion. The NC and AC hierarchies provide an interesting structural contrast to other hierarchies. In many respects they behave like the polynomial hierarchy. This is especially true when considering that for these hierarchies a collapse at one level spreads upward, and this collapse can be shown by a decomposition of the higher levels. Unlike the polynomial hierarchy, the NC and AC hierarchies are dense. In this, they act like the classical space/time complexity classes. This is reasonable: NC and AC are defined by allowing progressively more and more parallel time. For NC and AC, however, no separation results are known (aside from $AC_0 \neq NC_1$ [6]). A statement about the NC hierarchy that combines features of both the other hierarchies is the following:

> For any two rationals $r < t$ there exists an s such that $NC_r \subseteq NC_s \subseteq NC_t$ and if NC_s is equal to either NC_r or NC_t, then NC collapses at least to NC_s.

An interesting open question raised by Corollary 5.5 is the relationship between AC_a and NC_{a+1} reducibilities. For example, are they the same on L or NL: is $AC_a^L = NC_{a+1}^L$? Under what circumstances can we say that $A \leq^{NC_{a+1}} B$ implies that $A \leq^{AC_a} B$? Answering these questions should help us pinpoint the relationship of AC_a to NC_{a+1}.

In §7 we saw oracles A so that, for any rational k, $NC_k^A \subset AC_k^A$ and $AC_k^A \subset NC_{k+1}^A$. We would like to see oracles B and C where, for any k, $NC_k^B \subset AC_k^B = NC_{k+1}^B$ and $NC_k^C = AC_k^C \subset NC_{k+1}^C$. This would raise an intriguing possibility.

We conclude this paper by pointing out that although the method presented here may seem a natural way to provide NC and AC with an oracle, the corresponding problem for space-bounded classes has been much more difficult ([2],[9],[13]). This has been especially true when comparing relativized NC to relativized space.

Acknowledgments. Several people deserve my thanks for helping improve this paper. In particular, I have benefited from discussions with Ron Book, Ravi Boppana, Gene Luks, Larry Ruzzo, and Osamu Watanabe, as well as from the reports of the anonymous referees.

REFERENCES

[1] 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.

[2] J. BUSS, *Relativized alternation and space-bounded computation*, J. Comput. System Sci., 36 (1988), pp. 351–378.

[3] A. CHANDRA, L. STOCKMEYER, AND U. VISHKIN, *Constant depth reducibility*, SIAM J. Comput., 13 (1984), pp. 423–439.

[4] J. CHEN, *Logarithmic depth reducibility and the NC hierarchy*, unpublished manuscript, 1987.

[5] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.

[6] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial-time hierarchy*, Math. Systems Theory, 27 (1984), pp. 13–27.

[7] N. PIPPENGER, *On simultaneous resource bounds* (preliminary version), in Proc. 20th Annual IEEE Conference on Foundations of Computer Science, IEEE Press, 1979, pp. 307–311.

[8] W. L. RUZZO, *On uniform circuit complexity*, J. Comput. System Sci., 22 (1981), pp. 365–383.

[9] W. L. RUZZO, J. SIMON, AND M. TOMPA, *Space-bounded hierarchies and probabilistic computations*, J. Comput. System Sci., 28 (1984), pp. 216–230.

[10] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.

[11] H. VENKATESWARAN, *Properties that characterize* LOGCFL, in Proc. 19th Annual ACM Symposium on Theory of Computing, ACM Press, 1987, pp. 141–150.

[12] C. B. WILSON, *Relativized NC*, Math. Systems Theory, 20 (1987), pp. 13–29.

[13] ———, *A measure of relativized space which is faithful with respect to depth*, J. Comput. System Sci., 36 (1988), pp. 303–312.

SIAM J. COMPUT.
Vol. 19, No. 2, pp. 397–409, April 1990

012

PARALLEL DEPTH-FIRST SEARCH IN GENERAL DIRECTED GRAPHS*

ALOK AGGARWAL†, RICHARD J. ANDERSON‡, AND MING-YANG KAO§

Abstract. A directed cycle separator of an n-vertex directed graph is a vertex-simple directed cycle such that when the vertices of the cycle are deleted, the resulting graph has no strongly connected component with more than $n/2$ vertices. It is shown that the problem of finding a directed cycle separator is in randomized NC. It is also proved that computing cycle separators and conducting depth-first search in directed graphs are deterministically NC-equivalent. These two results together yield the first randomized NC algorithm for depth-first search in general directed graphs.

Key words. depth-first search, directed graphs, parallel algorithms, RNC, cycle separators

AMS(MOS) subject classifications. 68Q10, 05C99

1. Introduction.

Depth-first search is one of the most useful tools in graph theory [13], [2]. The *depth-first search problem* is the following: given a graph and a distinguished vertex, construct a tree that corresponds to performing depth-first search of the graph starting from the given vertex. In the setting of parallel computation, this problem has been studied by a number of authors. For lexicographic depth-first search, Reif shows that the problem is P-complete even for general undirected graphs [11]. Ghosh and Bhattacharjee provide an NC algorithm for lexicographic depth-first search in acyclic directed graphs [5]. Their algorithm has an error and is corrected by Zhang [14]. For unordered depth-first search, Smith gives the first NC algorithm for planar undirected graphs [12]. The processor complexity of his algorithm is reduced to linear by Ja'Ja and Kosaraju [8] and independently by He and Yesha [7]. Anderson provides an RNC algorithm to find a maximal path of a general undirected graph [3]; a maximal path is the first branch of a depth-first search tree. Aggarwal and Anderson give an RNC algorithm for general undirected graphs [1]. Kao provides a deterministic NC algorithm for planar directed graphs [9]. In this paper, we give the first RNC algorithm for *general directed* graphs.

Our general directed depth-first search algorithm uses a divide-and-conquer strategy similar to that used by Aggarwal and Anderson for general undirected depth-first search [1]. In addition to this strategy, a crucial idea used in the paper is that of *directed cycle separators.* This idea was originally introduced by Kao for planar directed depth-first search [9]. At the very highest level, our algorithm finds and removes a portion of a depth-first search tree of a given directed graph. The algorithm then recurses on strongly connected components as well as certain weakly connected subgraphs of the resulting graph. To limit the depth of recursion, directed cycle separators are used to divide the given graph into small pieces. While the undirected and directed depth-first search algorithms have similar structures, directed graphs require more

* Received by the editors September 12, 1988; accepted for publication (in revised form) August 22, 1989. A preliminary version of this paper appeared in the Proceedings of the 21st ACM Symposium on Theory of Computing, Seattle, Washington, May 15-17, 1989, pages 297-308.

† IBM Research Division, Thomas J. Watson Research Center, Box 218, Yorktown Heights, New York 10598.

‡ Department of Computer Science, University of Washington, Seattle, Washington 98195. The work of this author was supported in part by National Science Foundation under Presidential Young Investigator Award CCR-86-57562.

§ Department of Computer Science, Duke University, Durham, North Carolina 27706. The work of this author was performed while he was at the Department of Computer Science, Indiana University, Bloomington, Indiana 47405.

work. For instance, a major difference between the two algorithms arises in the construction of separators. Both of the algorithms construct separators by repeatedly joining paths until only a single path remains. However, a key idea in the undirected case is to carry out bisection by traversing and joining the longer half of a path; in the directed case, it is not possible to choose the direction of traversal. Therefore, joining directed paths requires a more sophisticated idea. Another major difference is in the application of the separator that allows the problem decomposition. In the undirected case, once a path separator is given, the decomposition of the graph is almost immediate. In the directed case, however, while the removal of a directed cycle separator makes all resulting strongly connected components reasonably small, the resulting graph may still have large weakly connected subgraphs. In order to successfully divide the graph for the recursive calls, certain weakly connected subgraphs also must be reduced, which requires a fair amount of work.

The parallel computation model used in this paper is the EREW PRAM model, i.e., no two processors are allowed to simultaneously read from or write into the same memory cell. Many of our complexity results are expressed in terms of $MM(n)$, which denotes the sequential time, currently $O(n^{2.376})$, for multiplying two $n \times n$ integer matrices in Strassen's model [2], [4]. This paper is organized as follows. Section 2 introduces the concept of directed cycle separators and discusses a number of preliminary results. Section 3 gives the first major result of this paper, establishing a deterministic NC-equivalence between finding directed cycle separators and performing directed depth-first search. Section 4 describes an NC reduction from the problem of finding a directed cycle separator to a particular kind of a matching problem. Section 5 combines these results together, estimates time complexity and processor bounds, and discusses open problems and extensions.

2. Directed cycle separators. A *separator* of a graph is a subgraph whose removal disconnects the graph into small pieces. Most of the works on parallel depth-first search rely on finding some form of graph separator. The general undirected depth-first search algorithm uses path separators [1]. The planar undirected depth-first search algorithms employ undirected cycle separators [12], [7], [8]. The planar directed depth-first search algorithm uses directed cycle separators and other kinds of separators in vertex-weighted graphs [9]. This paper follows the directed separator definition given by Kao [9]: a *separator* of an n-vertex directed graph G is a set of vertices S such that $G - S$ has no strongly connected component with more than $n/2$ vertices. A *directed path separator* is a vertex-simple directed path whose vertices form a separator; a *directed cycle separator* is a vertex-simple directed cycle whose vertices form a separator. A single vertex is considered a cycle of length zero; thus, if the removal of a vertex separates a graph, the vertex is a cycle separator. Kao has shown that every directed graph has a directed path separator and a directed cycle separator [9]. Furthermore, such a path separator is computable in linear sequential time, and such a cycle separator is computable within a $\log n$ factor of the optimal linear sequential time. Here we modify his proof and obtain the optimal sequential time bound.

THEOREM 2.1. *Every directed graph has a directed cycle separator. Such a separator can be found in $O(n+e)$ sequential time for any directed graph of n vertices and e arcs.*

Proof. Because every directed graph has a directed path separator and such a path separator can be found in linear time [9], it suffices to show that any directed path separator can be converted into a directed cycle separator in linear time. In the

following discussion we describe such a conversion in two steps.

Let G be a directed graph of n vertices. For a subgraph S and a vertex v in S, let $R_{in}(v, S)$ (or $R_{out}(v, S)$) denote the set of vertices that can reach (or respectively, can be reached from) v through directed paths in S. A directed path separator $P = x_1, \cdots, x_p$ is called *semiminimal* if $R_{out}(x_p, G - \{x_1, \cdots, x_{p-1}\})$ has more than $n/2$ vertices and $R_{in}(x_1, G - \{x_2, \cdots, x_p\})$ also has more than $n/2$ vertices. Given such a P, a directed cycle separator can be built in linear time as follows. There are two cases: $p = 1$ and $p > 1$. If $p = 1$, then x_1 alone forms a cycle separator. If $p > 1$, then because both $R_{in}(x_1, G - \{x_2, \cdots, x_p\})$ and $R_{out}(x_p, G - \{x_1, \cdots, x_{p-1}\})$ have more than $n/2$ vertices, the two sets share at least one common vertex. Consequently, there is vertex-simple directed path P' from x_p to x_1 such that P' is completely in the two sets. Because the two sets and P share only x_1 and x_p, P' and P form a vertex-simple directed cycle. Because P is already a separator, the cycle is a directed cycle separator. This step takes linear time because P' can be found in linear time.

To finish the proof, we show how to cut any directed path separator $Q = y_1, \cdots, y_q$ into a semiminimal one in linear time as follows. The idea is that if $R_{out}(y_q, G - \{y_1, \cdots, y_{q-1}\})$ has no more than $n/2$ vertices, then $Q' = y_1, \cdots, y_{q-1}$ is still a path separator. Otherwise, let C be the strongly connected component in $G - Q'$ such that C contains more than $n/2$ vertices. Because Q is a separator, C must contain y_q. This implies that C is a subset of $R_{out}(y_q, G - \{y_1, \cdots, y_{q-1}\})$, which is a contradiction. We can extend the above idea: if t is the largest index such that $R_{out}(y_t, G - \{y_1, \cdots, y_{t-1}\})$ has more than $n/2$ vertices, then $Q'' = y_1, \cdots, y_t$ is still a path separator. The index t can be identified easily in linear time by using the following recurrence formula. Let $R_{out,i}$ denote $R_{out}(y_i, G - \{y_1, \cdots, y_{i-1}\})$ for $i = 1, \cdots, q-1$. Then $R_{out,i} = R_{out,i+1} \cup R_{out}(y_i, G - R_{out,i+1} - \{y_1, \cdots, y_{i-1}\})$. After t is found, we perform the same computation on Q'' at the other end by computing R_{in}. After both ends of Q are processed, we have a semiminimal directed path separator. Since each end of Q can be cut in linear time, the whole process takes linear time. □

3. Using cycle separators to conduct depth-first search. Kao has also shown that given a directed depth-first search forest, finding a directed path separator is in deterministic NC, and given a directed path separator, finding a directed cycle separator is also in deterministic NC [9]. In this section, we will show that given an oracle for computing a directed cycle separator, conducting directed depth-first search is in deterministic NC. These results immediately imply the following theorem.

THEOREM 3.1. *For general directed graphs, computing a directed path separator, computing a directed cycle separator, and conducting directed depth-first search are deterministically NC-equivalent.*

We now discuss how to use directed cycle separators to conduct directed depth-first search in parallel. Suppose that we want to perform depth-first search in an n-vertex directed graph G starting from some vertex r. Any such search will visit exactly the vertices reachable from r using directed paths. We call a graph *rooted* at a vertex if the vertex can reach all other vertices through directed paths. We assume, for the moment, that G is rooted at r and our goal is to build a directed depth-first search spanning tree rooted at r for G. We will recursively construct, in parallel, such a tree using directed cycle separators.

We first explain why the straightforward recursive approach used in the undirected case [12], [1] does not work for directed graphs. Given a directed cycle separator, we can efficiently build a directed path separator P_r starting from r by finding a directed

path from r to the cycle separator. The path and the cycle separator form a directed path separator with a certain arc on the cycle removed. The path separator P_r will be a branch of the final depth-first search tree for the directed graph G. Now let G' be the remaining graph that is not searched by P_r, in other words, $G' = G - P_r$. Suppose that we continue to search G' starting from a vertex r' that is not in P_r but is the end vertex of an arc starting from the last vertex of P_r. This time we recurse on the subgraph $G'_{r'}$ that consists of all the vertices reachable from r' using directed paths in G'. Because P_r is a separator of G, every strongly connected component of G' has at most $n/2$ vertices. However, $G'_{r'}$ may contain several such strongly connected components. Consequently, $G'_{r'}$ may still be too large for small depth recursion. To avoid this problem, we describe below a more sophisticated subroutine that removes a set of directed paths from G such that the remaining directed graph has small rooted subgraphs. These removed paths will form a subtree in the final depth-first search tree.

A *partial* depth-first search tree in a rooted directed graph is a subtree of a depth-first search tree such that the graph and two trees are rooted at the same vertex. Let T be a partial depth-first search tree of G. Let $x_1, x_2, \cdots, x_t$ be the vertices of T listed in the post-order traversal sequence of depth-first search, i.e., in this sequence x_i is marked right after all its descendants in T are marked. For any x_i, let $y_{i,1}, \cdots, y_{i,k_i}$ be the vertices that are not in T but are the end vertices of the arcs starting from x_i. The order of $y_{i,1}, \cdots, y_{i,k_i}$ is arbitrary, and a y vertex may have several different indices if it is adjacent from several x vertices. For a directed graph D and a vertex $x \in D$, let $R(x, D)$ denote the set of vertices that can be reached from x using directed paths in D. We call a subgraph of G a *dangling subgraph*, denoted by $DSG((i,j), T)$, with respect to (i,j) and T if it is formed by the vertices in $G - T$ that can be reached from $y_{i,j}$ but not from $y_{i',j'}$ for any (i',j') such that either $i' < i$ or $(i' = i$ and $j' < j)$. In other words, $DSG((i,j), T) = R(y_{i,j}, G - T) - \cup\{R(y_{i',j'}, G - T) \mid$ either $i' < i$ or $(i' = i$ and $j' < j)\}$.

Observe that a depth-first search tree of G is simply the union of the arcs in T, the set of arcs $(x_i, y_{i,j})$ for which $DSG((i,j), T)$ is nonempty, and an arbitrary depth-first search tree rooted at $y_{i,j}$ for each nonempty $DSG((i,j), T)$. Also observe that because the dangling subgraphs are disjoint, we can simultaneously compute an arbitrary depth-first search tree for each nonempty dangling subgraph. These observations together provide a natural way to recursively and concurrently extend a partial depth-first search into a complete depth-first search tree.

To achieve small depth recursion, the nonempty dangling subgraphs must be small. Keeping this in view, we call a dangling subgraph *heavy* if it has more than $n/2$ vertices, otherwise we call it *light*. Similarly, we call a partial depth-first search tree *heavy* if it has a heavy dangling subgraph, otherwise we call it *light*. If a partial depth-first search tree is light, the recursion can be readily applied to its nonempty dangling subgraphs. So we may assume that the tree is heavy. Because all dangling subgraphs are vertex-disjoint, the tree has exactly one heavy dangling subgraph, denoted by $DSG((i_o, j_o), T)$. Let H denote the rooted acyclic directed graph induced by *contracting* the strongly connected components of $DSG((i_o, j_o), T)$. Furthermore, let a vertex in H be assigned the weight equal to the number of vertices in the corresponding strongly connected component. Since $DSG((i_o, j_o), T)$ is heavy and since H is acyclic and rooted, there exists a vertex in H such that the total weight of this vertex and its descendants is greater than $n/2$ but the weight of each of its children and the weights of this child's descendants sum up to at most $n/2$. Call such a ver-

tex a *splitting vertex*, and call the corresponding strongly connected component in $DSG((i_o, j_o), T)$ a *splitting component*. There may be several splitting vertices and splitting components. Now pick an *arbitrary* splitting vertex s, and denote its corresponding splitting component by G_s. Next use the given oracle to obtain a cycle separator C_s in G_s. Furthermore, build an arbitrary vertex-simple directed path P_o in $DSG((i_o, j_o), T)$ that goes from y_{i_o,j_o} to an arbitrary vertex of C_s and then traverses C_s except its last arc. P_o will be a branch of the final depth-first search tree of G; more precisely, let the new tree be $T' = T \cup \{(x_{i_o}, y_{i_o,j_o})\} \cup P_o$. Observe that T' is still a partial depth-first search tree of G. If T' has no heavy dangling subgraph, then we have achieved our goal of building a light partial depth-first search tree. So we may assume that T' has a heavy dangling subgraph.

LEMMA 3.2. *If T' is a heavy partial depth-first search tree, then every splitting component of T' is a strongly connected component of $G_s - P_o$, and consequently, consists of at most $|G_s|/2$ vertices.*

Proof. To locate the heavy dangling subgraph of T', observe that $DSG((x_o, y_o), T)$ is the union of $G_s \cap P_o$ and the dangling subgraphs of T' rooted at vertices that are both in $G_s - P_o$ and adjacent from $G_s \cap P_o$. This guarantees that the heavy dangling subgraph of T' is rooted at some vertex that is both in $G_s - P_o$ and adjacent from $G_s \cap P_o$. To further locate the splitting components of T', recall that from the definition, G_s corresponds to a vertex s in H such that the weight of each child of s and the weights of this child's descendants sum up to at most $n/2$. This implies that $G_s - P_o$ must include all splitting components of T'. Therefore, every splitting component of T' is a strongly connected component of $G_s - P_o$. Furthermore because P_o contains the cycle separator C_s of G_s, every splitting component of T' has at most $|G_s|/2$ vertices. □

From the above lemma, it is readily seen that after $O(\log n)$ such phases of cutting up splitting components, any resulting splitting component is left with a single vertex. If we now extend the partial depth-first search tree to include the vertex of an arbitrary splitting component, we can obtain another partial depth-first search tree T' that is light so that recursion can be performed.

We now summarize the above discussion. To construct a depth-first search tree, we start from the partial depth-first search tree T that consists of only the root r. It takes $O(\log n)$ cuts of splitting components to extend T into a light partial depth-first search tree. Each cut takes an oracle call to find a cycle separator in addition to $O(\log^2 n)$ time and $MM(n)$ processors for computing transitive closures of suitable graphs with at most n vertices. To build a complete depth-first search tree, we recurse on the nonempty dangling subgraphs of T. This recursion has depth $O(\log n)$. So if a directed cycle separator of an n-vertex strongly connected directed graph can be computed in $T_c(n)$ parallel time using $P_c(n)$ processors, then a depth-first search tree of any n-vertex rooted directed graph can be computed in $O(\log^2 n(T_c(n) + \log^2 n))$ time using $P_c(n) + MM(n)$ processors.

The above discussion applies only to rooted directed graphs. We now extend the discussion to any general directed graph G. Our goal is to find a depth-first search spanning forest with r being the first root. We first decompose G into rooted directed graphs as follows. Arrange the vertices of G in an arbitrary order $u_1, \cdots, u_n$ with $u_1 = r$. Let $D(u_i)$ be the set of vertices that can be visited by depth-first search starting from u_i but cannot be visited starting from $u_1, \cdots, u_{i-1}$; in other words, $D(u_i) = R(u_i, G) - \cup_{j<i} R(u_j, G)$. Clearly, each nonempty $D(u_i)$ is a directed graph rooted at u_i. Now a complete depth-first search of G starting from r can be conducted

by computing an arbitrary depth-first search tree for each nonempty $D(u_i)$ with the root being u_i. These nonempty $D(u_i)$'s can be computed in $O(\log^2 n)$ time and $MM(n)$ processors. Hence, from this discussion, we obtain the following theorem.

THEOREM 3.3. *Suppose a directed cycle separator of any n-vertex strongly connected directed graph can be computed in $T_c(n)$ time using $P_c(n)$ processors. Then a depth-first search spanning forest of any n-vertex general directed graph can be computed in $O(\log^2 n(T_c(n) + \log^2 n))$ time using $P_c(n) + MM(n)$ processors.*

4. Constructing a directed cycle separator. A *directed multipath separator* is a set of vertex-disjoint vertex-simple directed paths whose vertices form a separator. For obtaining a directed cycle separator of any n-vertex directed graph, in §4.1 we describe a routine REDUCE that given a directed multipath separator Ω of $2 \cdot (\lfloor \log n \rfloor + 2)$ paths or more, reduces the number of paths in Ω by at least one half so that the resulting set is still a directed multipath separator. Initially Ω is the directed multipath separator that consists of any $\lceil n/2 \rceil$ vertices in G, each vertex being a directed path of length zero. Since each call to REDUCE decreases the size of Ω to at most one half of its original size, $O(\log n)$ calls are sufficient to reduce the size of Ω to smaller than $2 \cdot (\lfloor \log n \rfloor + 2)$. Once Ω has fewer than $2 \cdot (\lfloor \log n \rfloor + 2)$ paths, we will merge the paths, one by one, into a single directed path separator, and then convert this path separator into a directed cycle separator. These last two steps are described in §4.2.

In the following discussion, the *lower segment* of a directed path $P = x_1, \cdots, x_k$ refers to the directed path $x_1, \cdots, x_{\lceil k/2 \rceil}$, and the *upper segment* of P refers to the directed path $x_{\lceil k/2 \rceil + 1}, \cdots, x_k$. For a set Ω of m directed paths, we use $|\Omega| = m$ to denote the number of paths in Ω. In unambiguous cases, we often refer to a directed path when, in fact, we mean the vertices of that directed path. For example, $G - P$ denotes the induced subgraph where all the vertices contained in P are removed, and $G - \Omega$ denotes the induced subgraph where all the vertices contained in the paths of Ω are removed.

4.1. Reducing the number of paths while maintaining the separator property. As highlighted in the Introduction, the basic idea used in the routine REDUCE is similar to that used by Aggarwal and Anderson for general undirected depth-first search [1]. The undirected depth-first search algorithm constructs a path separator by repeatedly joining paths until only a single path remains. A key idea in the undirected case is to carry out bisection by traversing and joining the longer half of a path. This bisection idea is not applicable in the directed case because it is not possible to choose the direction of traversal in directed paths. One of the new ideas for the directed case is that given two paths L and S, we find a directed path P that goes from L to the *lower* segment of S so that we can always traverse the longer half of S. To make this idea work, the directed REDUCE uses several other new ideas that are described in the following discussion.

The routine REDUCE takes as input a multipath separator Ω of m paths with $m \geq 2 \cdot (\lfloor \log n \rfloor + 2)$. REDUCE will operate for $O(\log^2 n)$ phases to reduce the size of Ω by at least one half so that the resulting set still forms a directed multipath separator. At the beginning of REDUCE, Ω is partitioned into two sets Δ and Γ. The set Γ is further partitioned into two sets, the set of *active* paths and that of *inactive* paths, denoted by Γ_a and Γ_{in}, respectively. These five sets are modified in the phases of REDUCE such that Ω has at most $m/2$ paths at the end of REDUCE. To be precise about the size of these sets, let $\alpha = \lfloor m/(2\lfloor \log n \rfloor + 4) \rfloor$ in the following discussion.

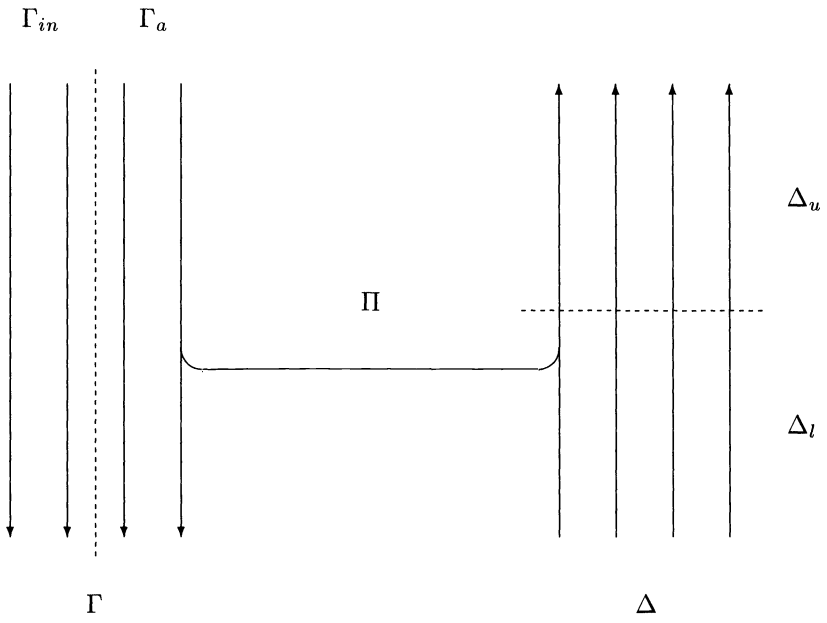


FIG. 1. Π *joins* Γ *and* Δ.

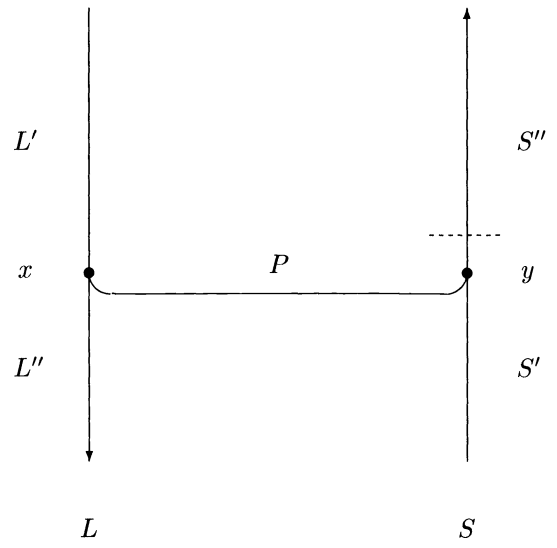


FIG. 2. *Rearrange two paths.*

- At the very beginning of REDUCE, $|\Gamma| = \alpha$, $\Gamma_a = \Gamma$, $|\Gamma_{in}| = 0$, and $|\Delta| = m - \alpha$. The assumption $|\Omega| = m \geq 2 \cdot (\lfloor \log n \rfloor + 2)$ is needed to ensure Γ is not empty, otherwise, REDUCE would never get started.
- At the very end of REDUCE, $|\Gamma_a| \leq \alpha$, $|\Gamma_{in}| \leq (\lfloor \log n \rfloor + 1) \cdot \alpha$, and $|\Delta| = 0$. Consequently, $|\Gamma| = |\Gamma_a| + |\Gamma_{in}| \leq (\lfloor \log n \rfloor + 2) \cdot \alpha$, and $|\Omega| = |\Gamma| + |\Delta| \leq (\lfloor \log n \rfloor + 2) \cdot \alpha$. The value of α is chosen to be $\lfloor m/(2\lfloor \log n \rfloor + 4) \rfloor$ so that $|\Omega| \leq m/2$.

To achieve this reduction, in each phase we find a set of paths Π that joins paths in Ω in a suitable manner. (See Fig. 1.) More precisely, let Δ_u and Δ_l denote the sets of, respectively, upper segments and lower segments of the paths in Δ. Π is a maximal set of vertex-disjoint vertex-simple directed paths that go from Γ_a to Δ_l such that for each directed path P in Π, the first vertex belongs to a path in Γ_a, the last vertex belongs to a path in Δ_l, and the interior vertices are taken from $G - \Omega$. Each directed path in $\Gamma_a \cup \Delta_l$ contains an end vertex of at most one path of Π. Of course, there may be directed paths in $\Gamma_a \cup \Delta_l$ that do not contain any end vertices of the paths in Π. Below we describe the basic step in using Π to rearrange paths in Ω. (See Fig. 2.) Suppose that a directed path $P \in \Pi$ joins directed paths $L \in \Gamma_a$ and $S \in \Delta$, and that P has end vertices x and y where $L = L'xL''$ and $S = S'yS''$. In each phase of REDUCE, L is replaced by $L'PS''$, and S is replaced by S'. The path L'' is either added to Γ_{in} or is discarded from Ω. The conditions under which L'' is discarded from Ω and those under which L'' is added to Γ_{in} will be discussed later. Irrespective of whether L'' is discarded from Ω or added to Γ_{in}, note that the directed path S has been reduced to half its original length because S' is only a subpath of the lower segment of the original S. Furthermore, the paths in Γ_a and those in Δ do not increase in number; in fact, the number of paths in Γ_a and Δ may have decreased if some paths of Π have been joined to the lowest end vertices in Δ_l. However, the number of paths in Γ_{in} may increase, which may, in turn, lead to an increase in the total number of paths in $\Gamma = \Gamma_a \cup \Gamma_{in}$. If the size of Δ does not decrease, then the increase in $|\Gamma|$ can increase the number of paths in $\Omega = \Gamma \cup \Delta$. We have mentioned above that Π is a maximal set, and in the following discussion, we will specify another property of Π that will help in eventually reducing the number of directed paths in Ω rather than increasing it.

We introduce two kinds of notation to describe a more detailed picture of how Π is used to reduce the cardinality of Ω (see Fig. 3): (1) $\Gamma_{a,new}$ denotes the set of all paths of the kind $L'PS''$. Γ_a^* denotes the set of all paths of the kind L''. Δ_{new} denotes the set of all paths of the kind S'. (2) $\hat{\Gamma}_a$ denotes the set of all paths in Γ_a that are not connected by any paths in Π to any paths in Δ_l. $\hat{\Delta}$ denotes the set of all paths in Δ whose lower segments are not connected by any paths in Π to any paths in Γ_a. Also $\hat{\Delta}_l$ and $\hat{\Delta}_u$ denote the sets of, respectively, lower segments and upper segments of paths in $\hat{\Delta}$. We can now specify the additional property for Π: Π *is such that there are no directed paths from* $\hat{\Gamma}_a \cup \Gamma_a^*$ *to* $\hat{\Delta}_l$ *using vertices in* $G - \Omega - \Pi$. Clearly Π is a maximal set and we call it a *maximal joining set from* Γ *to* Δ. Later in this section we will discuss how to compute such Π. Here we continue to explain how Π is used to reduce the cardinality of Ω. Let $\mathrm{scc}(D)$ denote the size of the largest strongly connected component in any directed graph D. The properties of Π imply that $\Gamma_a^* \cup \hat{\Gamma}_a$ and $\hat{\Delta}_l$ cannot have vertices in the same strongly connected component of $(G - \Omega - \Pi) \cup (\Gamma_a^* \cup \hat{\Gamma}_a) \cup \hat{\Delta}_l$. This and the fact that $\Pi \cup \Omega$ is a separator of G in turn imply either $\mathrm{scc}((G - \Omega - \Pi) \cup (\Gamma_a^* \cup \hat{\Gamma}_a)) \leq n/2$, or $\mathrm{scc}((G - \Omega - \Pi) \cup \hat{\Delta}_l) \leq n/2$, or both. Based on these bounds, we have two cases for updating the sets Δ, Γ_a, and

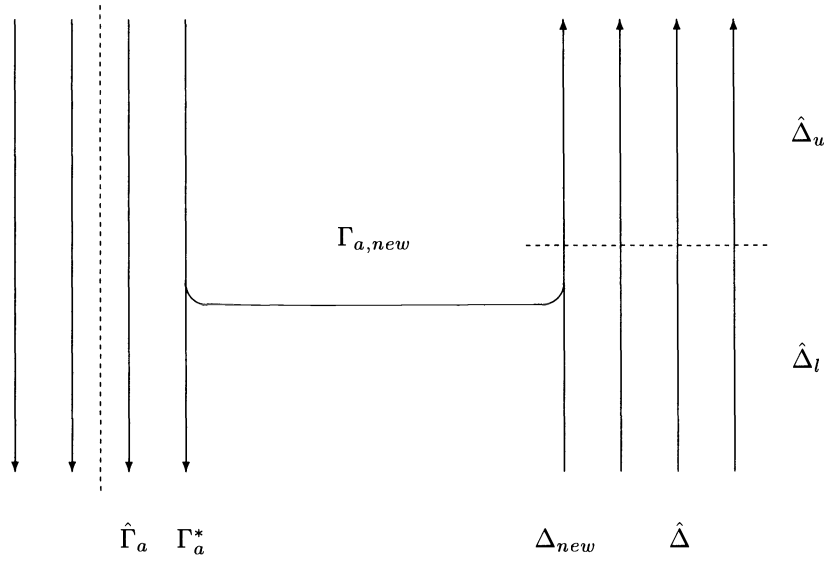


FIG. 3. *Notation for the subroutine* REDUCE.

Γ_{in}; the new version of these sets will be used in the next phase of REDUCE.

Case 1. If $\text{scc}((G - \Omega - \Pi) \cup \hat{\Delta}_l) \leq n/2$, we add Π to Ω but discard $\hat{\Delta}_l$ from Ω, and observe that the new Ω is still a directed multipath separator. Moreover, we perform the following replacements:

- $\Delta \leftarrow \Delta_{new} \cup \hat{\Delta}_u$,
- $\Gamma_a \leftarrow \Gamma_{a,new} \cup \hat{\Gamma}_a$, and
- $\Gamma_{in} \leftarrow \Gamma_{in} \cup \Gamma_a^*$.

Case 2. If $\text{scc}((G - \Omega - \Pi) \cup (\Gamma_a^* \cup \hat{\Gamma}_a)) \leq n/2$, we add Π to Ω but discard $\Gamma_a^* \cup \hat{\Gamma}_a$, and note that the new Ω is still a directed multipath separator. The number of paths in Γ_a may drop below α, which is the size of Γ_a at the very beginning of REDUCE. To restore this size, we take enough paths from $\Delta_{new} \cup \hat{\Delta}$ and add them to Γ_a until Γ_a is restored to its original cardinality or $\Delta_{new} \cup \hat{\Delta}$ is exhausted. Let Δ_a denote the set of paths taken from $\Delta_{new} \cup \hat{\Delta}$. Now we employ the following replacements:

- $\Delta \leftarrow \Delta_{new} \cup \hat{\Delta} - \Delta_a$,
- $\Gamma_a = \Gamma_{a,new} \cup \Delta_a$, and
- Γ_{in} remains unchanged.

(See Fig. 4 for a brief summary of the routine REDUCE.)

LEMMA 4.1. *If we have a directed multipath separator of m paths with $m \geq 2 \cdot (\lfloor \log n \rfloor + 2)$, then after executing $O(\log^2 n)$ phases of* REDUCE, *we can obtain a directed multipath separator with at most $m/2$ paths.*

Proof. First of all, observe that Ω remains a directed multipath separator after the replacements in each phase. Consequently, at the end of the routine, Ω is still a directed multipath separator. To reduce the size of Ω, REDUCE is executed until Δ becomes empty. In the following discussion, we will first discuss the situations under which Δ gets exhausted. We will then estimate $|\Gamma_a|$, $|\Gamma_{in}|$, and $|\Omega| = |\Gamma_a| + |\Gamma_{in}| + |\Delta|$ as at the end of REDUCE.

routine REDUCE(Ω):
  Partition Ω into Δ and Γ, and further partition Γ into Γ_a and Γ_{in};
  while Δ is not empty **do**
    begin
      Find a maximal joining set Π from Γ to Δ;
      If $\text{scc}((G - \Omega - \Pi) \cup \hat{\Delta}_l) \leq n/2$
        then
          begin
            $\Delta \leftarrow \Delta_{new} \cup \hat{\Delta}_u$;
            $\Gamma_a \leftarrow \Gamma_{a,new} \cup \hat{\Gamma}_a$;
            $\Gamma_{in} \leftarrow \Gamma_{in} \cup \Gamma_a^*$
          end
        else {comment: $\text{scc}((G - \Omega - \Pi) \cup (\Gamma_a^* \cup \hat{\Gamma}_a)) \leq n/2$}
          begin
            $\Delta \leftarrow \Delta_{new} \cup \hat{\Delta} - \Delta_a$;
            $\Gamma_a = \Gamma_{a,new} \cup \Delta_a$;
            Γ_{in} remains unchanged
          end
    end

FIG. 4. *A brief summary of* REDUCE.

We divide Case 2 given above into two cases. Case 2a: $|\Gamma_a^*| \geq \frac{1}{2} \cdot \alpha$ (equivalently, $|\hat{\Gamma}_a| < \frac{1}{2} \cdot \alpha$). Case 2b: $|\Gamma_a^*| < \frac{1}{2} \cdot \alpha$ (equivalently, $|\hat{\Gamma}_a| \geq \frac{1}{2} \cdot \alpha$). Also, let t_1, t_{2a}, and t_{2b}, respectively, denote the numbers of phases in which Cases 1, 2a, and 2b occur. In all three cases, $|\Delta|$ never increases and may sometimes decrease. This decrease happens when a path of Δ is taken to replenish Γ_a in Cases 2a and 2b, or when a path of Δ is cut sufficiently many times in Cases 1, 2a, and 2b. If a path is cut, it is cut by at least one half because it is replaced either by a subpath of its lower segment or by its upper segment. There are two situations in which Δ may become empty. One situation is that the paths of Δ are primarily cut off. Because a path has at most n vertices, it can allow at most $\lfloor \log n \rfloor + 1$ cuts. After these cuts, the path becomes empty. In Case 1, all paths in Δ are cut; in Case 2a, at least $\frac{1}{2} \cdot \alpha$ paths are cut; in Case 2b, the number is insignificant for our analysis. Because originally $|\Delta| = m - \alpha$, after $t_1 + t_{2a} + t_{2b}$ phases, Δ is left with a reserve of at most C_1 cuts where $C_1 = (m - \alpha) \cdot (\lfloor \log n \rfloor + 1) - t_1 \cdot (m - \alpha) - t_{2a} \cdot (\frac{1}{2} \cdot \alpha)$. Some of the paths in Δ may be taken away to replenish Γ_a in Cases 2a and 2b. The effect of this happening is canceled out in the estimate. Now, if C_1 is negative or zero, then Δ must be empty. C_1 can be negative or zero if $t_1 \geq \lfloor \log n \rfloor + 1$ (referred to as condition 1) or if $t_1 < \lfloor \log n \rfloor + 1$ but $t_{2a} \geq 4 \cdot (\lfloor \log n \rfloor + 2) \cdot (\lfloor \log n \rfloor + 1)$ (referred to as condition 2a). The other situation is that the paths of Δ are primarily taken away to replenish Γ_a in Cases 2a and 2b. The number of paths moved from Δ to Γ_a is $|\hat{\Gamma}_a|$. In Case 2b, this number is at least $\frac{1}{2} \cdot \alpha$; in Case 2a, this number is insignificant. After $t_1 + t_{2a} + t_{2b}$ phases, $|\Delta|$ is at most $C_2 = (m - \alpha) - t_{2b} \cdot (\frac{1}{2} \cdot \alpha)$. So if C_2 is negative or zero, then Δ must be empty. C_2 can be negative or zero if $t_1 < \lfloor \log n \rfloor + 1$ but $t_{2b} \geq 4 \cdot (\lfloor \log n \rfloor + 2)$ (referred to as condition 2b).

Now we estimate $|\Gamma_a|$, $|\Gamma_{in}|$, $|\Gamma| = |\Gamma_a| + |\Gamma_{in}|$, and $|\Omega| = |\Gamma| + |\Delta|$. Originally $|\Gamma_a| = \alpha$. $|\Gamma_a|$ does not change in Case 1 and does not increase in Cases 2a and 2b. So $|\Gamma_a| \leq \alpha$ throughout the execution of REDUCE. Because $|\Gamma_{in}|$ may increase by at

most $|\Gamma_a| \leq \alpha$ in Case 1 and does not change in Cases 2a and 2b, after $t_1 + t_{2a} + t_{2b}$ phases, $|\Gamma_{in}| \leq t_1 \cdot \alpha$ and consequently, $|\Gamma| \leq \alpha + t_1 \cdot \alpha$. Furthermore, $|\Gamma| \leq m/2$ if any one of the conditions 1, 2a, or 2b is true. REDUCE can achieve at least one of these three conditions within $(\lfloor \log n \rfloor + 1) + \{4 \cdot (\lfloor \log n \rfloor + 2) \cdot (\lfloor \log n \rfloor + 1)\} + \{4 \cdot (\lfloor \log n \rfloor + 2)\}$ phases. Hence, within $O(\log^2 n)$ phases, $|\Omega| = |\Gamma| \leq m/2$. □

To complete the description of REDUCE, we explain how to compute Π as follows. Recall that Π must be a maximal joining set from Γ to Δ. Here we give a stronger property for Π. For the previously described path P in the set Π, we assign a cost equal to the length of the cut-off segment of L, namely, the number of vertices in L''. We call Π a *minimum-cost maximum-cardinality joining set* if Π has the maximum number of paths from Γ_a to Δ_l, and if Π also minimizes the total cost under the maximum-cardinality constraint. Below we prove that if Π is a minimum-cost maximum-cardinality joining set from Γ to Δ, then Π is a maximal joining set from Γ to Δ: Because Π is of the maximum-cardinality, there can be no directed path from $\hat{\Gamma}_a$ to $\hat{\Delta}_l$ using vertices in $G - \Omega - \Pi$; if there were such a path, this path could be added to Π and the original Π would not be of the maximum cardinality. Because Π is also of the minimum total cost, there can be no directed path from Γ_a^* to $\hat{\Delta}_l$ using vertices in $G - \Omega - \Pi$; if there were such a path, then a certain path of Π could be replaced by this smaller cost path while Π maintained the same maximum cardinality.

In view of the above discussion, to find a maximal joining set Π from Γ to Δ, we only need to find a minimum-cost maximum-cardinality joining set from Γ to Δ. Aggarwal and Anderson have shown how to reduce an undirected version of the problem of finding a minimum-cost maximum-cardinality joining set to that of finding a minimum-weight perfect matching in a bipartite graph [1]. Essentially the same reduction can be used to solve our problem of finding Π. The reader is referred to their paper for details. Here we simply state the result as follows. Let $P_{mm}(n)$ and $T_{mm}(n)$ denote the number of processors and the parallel time to compute a minimum-weight perfect matching of any n-vertex bipartite graph that has an integer weight of at most n on each of its arcs. Then a maximal joining set Π from Γ to Δ can be found in exactly the same complexity. We summarize the discussion of this section in the following theorem.

THEOREM 4.2. *Let $P_{mm}(n)$ and $T_{mm}(n)$ denote the number of processors and the parallel time to compute a minimum-weight perfect matching of any n-vertex bipartite graph that has an integer weight of at most n on each of its arcs. Then the routine* REDUCE *can be used to obtain a directed multipath separator of fewer than $2 \cdot (\lfloor \log n \rfloor + 2)$ paths in $O(\log^3 n \cdot (T_{mm}(n) + \log^2 n))$ time using $P_{mm}(n) + MM(n)$ processors.*

Proof. Initially we have the directed multipath separator Ω that consists of any $\lceil n/2 \rceil$ vertices in G, each vertex being a directed path of length zero. From Lemma 4.1, if $|\Omega| \geq 2 \cdot (\lfloor \log n \rfloor + 2)$, then a call to REDUCE cuts $|\Omega|$ by at least one half. Therefore, $O(\log n)$ calls to REDUCE are sufficient to obtain a directed multipath separator of the desired size. Each call has $O(\log^2 n)$ phases. Each phase does two major computations: (1) computing a maximal joining set Π, and (2) computing the strongly connected components of an n-vertex directed graph in $O(\log^2 n)$ time using $MM(n)$ processors. So the total complexity is $O(\log^3 n \cdot (T_{mm}(n) + \log^2 n))$ parallel time and $P_{mm}(n) + MM(n)$ processors. □

4.2. Constructing a cycle separator from a small set of separating paths. Given a directed multipath separator Ω with fewer than $2 \cdot (\lfloor \log n \rfloor + 2)$ paths, we explain below how to construct a directed path separator and convert this path sep-

arator into a directed cycle separator. The idea is to repeatedly join two paths of Ω into a new path, whereas the other paths stay untouched and, together with the new path, remain a directed multipath separator. To join two paths, we first recall that Kao has given an NC algorithm to convert any directed path separator into a directed cycle separator [9]. In fact, the proof of Theorem 2.1 can also be used to do the conversion in parallel. The idea of joining two directed paths into one is almost the same as converting a directed path separator into a directed cycle separator. The only difference is that in the path-to-cycle conversion we merge the two ends of the given directed path separator, whereas in the path-to-path conversion we merge the ends of two paths, one end from each path. More precisely, let $\Omega = \{P_1, \cdots, P_k\}$ with $k < 2 \cdot (\lfloor \log n \rfloor + 2)$; also let $\Omega' = \{P_3, \cdots, P_k\}$. Below, we describe how to merge P_1 and P_2 into a single path P' so that P' and Ω' form a multipath separator. We will process P_1 and P_2 in essentially the same way as the proof of Theorem 2.1. Let $P_1 = x_1, \cdots, x_{p_1}$. Find the largest index t such that $R_{out}(x_t, G - \{x_1, \cdots, x_{t-1}\} - P_2 - \Omega')$ has more than $n/2$ vertices. If t does not exist, then let $P' = P_2$ and observe that P' and Ω' form a multipath separator. If t exists, then let $P_1' = x_1, \cdots, x_t$ and note that P_1', P_2, and Ω' form a multipath separator. Now let $P_2 = y_1, \cdots, y_{p_2}$. Find the smallest index s such that $R_{in}(y_s, G - \{y_{s+1}, \cdots, y_{p_2}\} - P_1' - \Omega')$ has more than $n/2$ vertices. If s does not exist, then let $P' = P_1'$ and observe that P' and Ω' form a multipath separator. If s exists, then let $P_2' = y_s, \cdots, y_{p_2}$, and observe that P_1', P_2', and Ω' form a multipath separator. Now find a vertex-simple directed path Q from x_t to y_s with internal vertices belonging to $G - P_1' - P_2' - \Omega'$. Let P' be the path formed by P_1', Q, and P_2'. It is readily seen that P' and Ω' form a multipath separator.

After repeating the above process $k - 1$ times, we can obtain a directed path separator. We then convert this path separator into a cycle separator. The complexity for merging two paths or converting a path into a cycle is $O(\log^2 n)$ time and $MM(n)$ processors because the only major computation in the merge is to find transitive closure of an appropriate directed graph. Moreover, because Ω has $O(\log n)$ paths, the total complexity of merging Ω into a directed cycle separator is $O(\log^3 n)$ time and $MM(n)$ processors. This discussion and Theorem 4.2 immediately yield the following theorem.

THEOREM 4.3. *Let $P_{mm}(n)$ and $T_{mm}(n)$ denote the processor and time complexities for computing a minimum-weight perfect matching of any n-vertex bipartite graph with an integer weight of most n on each arc. Then a directed cycle separator of any general n-vertex directed graph can be found in $O(\log^3 n(T_{mm}(n) + \log^2 n))$ time using $P_{mm}(n) + MM(n)$ processors.*

5. Discussions. Using Theorems 3.3 and 4.3, we can now state the other main results of this paper.

THEOREM 5.1. *Let $T_{mm}(n)$ and $P_{mm}(n)$ denote the parallel time and the number of processors to compute a minimum-weight perfect matching of any n-vertex bipartite graph with an integer weight of at most n on each arc. Furthermore, let $MM(n)$ denote the sequential time complexity of multiplying two $n \times n$ integer matrices in Strassen's model. Then a depth-first search forest of any general n-vertex directed graph can be computed in $O(\log^5 n(T_{mm}(n) + \log^2 n))$ time using $P_{mm}(n) + MM(n)$ processors.*

Mulmuley, Vazirani, and Vazirani give an RNC minimum-weight perfect matching algorithm such that $T_{mm}(n) = O(\log^2 n)$ and $P_{mm}(n) = n \cdot MM(n)$ [10]. Consequently, the above theorem has the following implication.

THEOREM 5.2. *For any general n-vertex directed graph, a depth-first search forest*

can be constructed probabilistically in $O(\log^7 n)$ *time using* $n \cdot MM(n)$ *processors.*

Finally, we conclude the paper with two challenging open problems. One problem is to devise a deterministic NC algorithm for general directed depth-first search. For the time being, we can only show that general directed depth-first search can be conducted deterministically in $O(\log^{11} n \cdot \sqrt{n})$ time using $O(n^3)$ processors. The result is obtained from modifying an undirected maximal joining set algorithm by Goldberg, Plotkin, and Vaidya [6]. The other problem is to find a more efficient RNC algorithm for general directed depth-first search. Because depth-first search is extremely useful in graph theory, an RNC algorithm with almost linear processor-time product will have significant impacts.

REFERENCES

[1] A. AGGARWAL AND R. ANDERSON, *A random* NC *algorithm for depth first search*, Combinatorica, 8 (1988), pp. 1–12.

[2] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[3] R. J. ANDERSON, *A parallel algorithm for the maximal path problem*, Combinatorica, 7 (1987), pp. 315–326.

[4] D. COPPERSMITH AND S. WINOGRAD, *Matrix multiplication via arithmetic progressions*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 1–6.

[5] R. K. GHOSH AND G. P. BHATTACHARJEE, *A parallel search algorithm for directed acyclic graphs*, BIT, 24 (1984), pp. 134–150.

[6] A. GOLDBERG, S. PLOTKIN, AND P. VAIDYA, *Sublinear-time parallel algorithms for matching and related problems*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 174–185.

[7] X. HE AND Y. YESHA, *A nearly optimal parallel algorithm for constructing depth first spanning trees in planar graphs*, SIAM J. Comput., 17 (1988), pp. 486–491.

[8] J. JA'JA AND S. KOSARAJU, *Parallel algorithms for planar graphs and related problems*, IEEE Trans. Circuits and Systems, 35 (1988), pp. 304–311.

[9] M. Y. KAO, *All graphs have cycle separators and planar directed depth-first search is in* DNC, in Proc. 3rd Aegean Workshop on Computing, Corfu, Greece, J. H. Reif, ed.; Lecture Notes in Computer Science 319, Springer-Verlag, Berlin, New York, 1988, pp. 53–63.

[10] K. MULMULEY, U. VAZIRANI, AND U. VAZIRANI, *Matching is as easy as matrix inversion*, Combinatorica, 7 (1987), pp. 105–114.

[11] J. H. REIF, *Depth-first search is inherently sequential*, Inform. Process. Lett., 20 (1985), pp. 229–234.

[12] J. R. SMITH, *Parallel algorithms for depth first search* I. *Planar graphs*, SIAM J. Comput., 15 (1986), pp. 814–830.

[13] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.

[14] Y. ZHANG, *Parallel algorithms for problems involving directed graphs*, Ph.D. thesis, Drexel University, Philadelphia, PA, 1986.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 411-423, June 1990

EXPRESSIBILITY AND NONUNIFORM COMPLEXITY CLASSES*

B. MOLZAN†

Abstract. A framework that admits the characterization of nonuniform complexity classes in terms of logical expressibility is presented.

In the case of classes that are defined by means of bounded-depth Boolean circuits, group theoretic characterizations given by Barrington and Thérien (see [*Proc. 19th Annual* ACM *Symposium on Theory of Computing*, Association for Computing Machinery, New York, 1987, pp. 101-109]) are exploited. This approach extends previous results of Immerman [*Proc. 15th Annual* ACM *Symposium on Theory of Computing*, Association for Computing Machinery, New York, 1983, pp. 347-354] and Gurevich and Lewis [*Inform. and Control*, 61 (1984), pp. 65-74], and provides a unique fashion of proofs for the results.

In the last section the computing power of generalized branching programs is related to logical expressibility.

Key words. computational complexity, nonuniform complexity classes, logical expressibility, AC^0-circuits, NC^1-circuits, branching programs

AMS(MOS) subject classifications. 68C25, 03C80

1. Introduction.

Usually, a *computational problem* is given by a finite alphabet and a language over this alphabet. Moreover, we can confine ourselves to languages over $\{0, 1\}$.

Equivalently, we may think of a problem as a class of finite structures of a finite relational signature. Doing so, it is natural to ask whether a complexity class can be characterized by means of logical invariants of its structure classes. It turns out that, in many cases, the expressive tools used to define the structure classes are such invariants.

This approach to complexity theory originates in [5], where it has been proved that a problem belongs to *NP* if and only if it can be expressed by means of existential second-order logic. Much work has been done to characterize other complexity classes in an analogous way. The papers [8] and [11] are good surveys on the results obtained so far.

In this paper we are concerned with so-called *nonuniform* complexity classes. Among the various definitions of *nonuniform computation*, the striking point is that a completely separate computational device (finite automaton, Turing machine, advice in addition to a Turing machine, Boolean circuit, etc.) may be used for each input size.

Corresponding to this notion of computation, nonuniformity must be reflected by the logical description of problems belonging to a complexity class. For this reason we introduce a logical framework appropriate for our purposes, admitting a logical characterization of the nonuniform complexity classes *NU-LogSpace*, *NU-NLogSpace*, and *NU-PTime* as well as of the circuit complexity classes AC^0, $AC^0[MOD_q]$, and NC^1. Finally, we show that some results on generalized branching programs can be proved by means of our characterization results.

The paper is organized as follows. Section 2 provides the logical tools. In § 3 the results are presented, whereas §§ 4 and 5 are devoted to the proofs.

2. Tools from logic.

2.1. Throughout this paper, by a *logic* we will understand a system $\mathcal{L}$, defined by the following:

* Received by the editors July 21, 1988; accepted for publication (in revised form) May 3, 1989.

† Akademie der Wissenschaften, Karl-Weierstrass-Institut für Mathematik, Mohrenstrasse 39, 1086 Berlin, German Democratic Republic.

(1) A class of symbols that, independently of a concrete signature, may be used to build formulas. (We will refer to these symbols as *logical symbols.*)

(2) Syntactical rules to construct formulas with respect to a given signature σ. (We will confine ourselves to signatures consisting of relation and constant symbols only.)

(3) For each σ, a class of *structures* of this signature.

(4) A *semantics*, i.e., a relation

$$\mathcal{A} \models_{\mathcal{L}} \varphi,$$

where $\mathcal{A}$ is an $\mathcal{L}(\sigma)$-structure and φ an $\mathcal{L}(\sigma)$-sentence (= formula where all variables are bounded by quantifiers).

2.2. We will regard finite structures only. To the logical symbols we always include the binary relation symbols $<$ and $=$ as well as the constant symbols min and max. We require that in any structure $<$ and $=$ are to be interpreted as a linear ordering and the identity on the universe, respectively. min and max will be interpreted as the least and greatest element as regards the ordering $<$. Thus we may assume that our universes are the sets $[n] = \{1, \cdots, n\}$ with the natural ordering, up to isomorphism.

Combining these conventions with the usual syntax and semantics of first-order logic, we obtain a variant of first-order logic for finite structures. We will denote it by *FO*.

To avoid trivial cases, our general convention on the cardinality of structures will be $n \geqq 2$. We extend this convention to the input size of computational problems, i.e., from now on we will regard finite structures of cardinality greater than or equal to two and computational problems for inputs of length greater than or equal to two only.

2.3. We will extend *FO* by new logical symbols, the interpretation of which depends only on the cardinality of a given structure. This idea is related to the concept of nonuniform computation, where a separate computational device may be used for each input size.

A *k-ary nonuniform relation* over $\mathbf{N}$ is a sequence $R = (R_n)_{n \in \mathbf{N}}$ of relations $R_n \subseteq [n]^k$.

Similarly, we define a *nonuniform constant* to be a sequence $c = (c_n)_{n \in \mathbf{N}}$ of natural numbers $c_n \in [n]$.

Now we can determine the components of a logic FO_{nu} as follows:

(1) In addition to the symbols of *FO* (e.g., logical connectives, quantifiers, variables, brackets, $<$ and $=$) we introduce a k-ary relation symbol $\mathbf{R}$ for each k-ary nonuniform relation R, and a constant symbol $\mathbf{c}$ for each nonuniform constant c.

(2) Given a relational signature σ, formulas are built from these symbols as in first-order logic.

(3) An $FO_{nu}(\sigma)$-structure consists, up to isomorphism, of a universe $[n]$ ($n \in \mathbf{N}$) together with an appropriate interpretation of σ.

(4) Given an $FO_{nu}(\sigma)$-structure $\mathcal{A}$ over $[n]$ and an $FO_{nu}(\sigma)$-sentence φ, we interpret the symbols for nonuniform relations and constants possibly occurring in φ by the corresponding relations and constants over $[n]$ (R_n for $\mathbf{R}$, c_n for $\mathbf{c}$), and then we define

$$\mathcal{A} \models_{FO_{nu}} \varphi$$

as in first-order logic.

2.4. It is well known that the expressive power of first-order logic can be increased by adding the *least fixed point operator*, thus permitting the formalization of inductive

definitions. The importance of this concept with regard to complexity theory is pointed out, for example, in [8]. The basic idea is as follows.

Assume that φ is a formula of the signature $\sigma \cup \{P\}$, where P is a new k-ary predicate symbol not occurring in σ, and let $x_1, \cdots, x_k$ be the free variables in φ. For each σ-structure, $\mathscr{A} = ([n], \cdots)$ φ defines an operator

$$\pi_\varphi : P([n]^k) \to P([n]^k),$$

$$R \subseteq [n]^k \mapsto \{\hat{a} \in [n]^k \mid (\mathscr{A}, R) \models \varphi(\hat{a})\}.$$

Moreover, if the new predicate symbol P occurs only positively in φ (i.e., under an even number of negation symbols), then this operator π_φ is monotone. Hence it has a least fixed point which will be denoted by

$$LFP^{\mathscr{A}}_{P,\hat{x}}[\varphi].$$

It is obvious how to add this construct to the syntactical and semantical rules of first-order logic, thus obtaining a new type of formula:

$$(y_1, \cdots, y_k) \in LFP_{P,\hat{x}}[\varphi].$$

The *transitive closure operator* may be viewed as a special case of the least fixed point operator. Given a formula $\varphi(\hat{x}, \hat{y})$, where $\hat{x}, \hat{y}$ are k-tuples of free variables,

$\mathscr{A} \models (\hat{a}, \hat{b}) \in TC_{\hat{x},\hat{y}}[\varphi]$ if and only if $(\hat{a}, \hat{b})$ belongs to the reflexive, transitive closure of the $2k$-ary relation defined by φ over the universe of $\mathscr{A}$.

We use $(\hat{u}, \hat{v}) \in dTC_{\hat{x},\hat{y}}[\varphi]$ as an abbreviation of

$$(\hat{u}, \hat{v}) \in TC_{\hat{x},\hat{y}}[\varphi(\hat{x}, \hat{y}) \wedge \forall \hat{z}[\varphi(\hat{x}, \hat{z}) \to \hat{z} = \hat{y}]],$$

thus obtaining the operator of *deterministic transitive closure.*

The expressive power of these and some other variants of the least fixed point operator with respect to complexity theory has been studied intensively (see, for example, [13]).

2.5. Given a logic $\mathscr{L}$, we will regard mainly $\mathscr{L}(H)$, the instance of $\mathscr{L}$ on a signature with a single unary predicate symbol H. Clearly, any interpretation of H over $[n]$ corresponds in a canonical way to a word $w \in \{0,1\}^n$ and vice versa.

We call a language $L \subseteq \{0,1\}^*$ $\mathscr{L}(H)$-*definable* if and only if there exists an $\mathscr{L}(H)$-sentence φ such that for all $w \in \{0,1\}^*$

$$w \in L \Leftrightarrow ([|w|], H_w) \models_{\mathscr{L}} \varphi.$$

We will apply this notion of definability to logics $\mathscr{L}$ that are obtained from FO_{nu} by adding various forms of the least fixed point operator, such as, for example, the following: $(FO_{nu} + dTC)$, $(FO_{nu} + TC)$, etc.

3. Presentation of the results.

3.1. We start our discussion with a characterization of the classical (uniform) complexity classes *LogSpace*, *NLogSpace*, and *PTime*, consisting of all problems that are computable in logarithmic space, nondeterministic logarithmic space, and polynomial time, respectively.

For these classes the following results have been proved:

(1) $LogSpace = \{K \subseteq Struc(\sigma) \mid \sigma$ is a finite relational structure and K is definable by an $(FO + dTC)(\sigma)$-*sentence*$\}$ [10].

(2) $NLogSpace = \{K \subseteq Struc(\sigma) \mid \sigma$ is a finite relational structure and K is definable by an $(FO + TC)(\sigma)$-*sentence*$\}$ ([10] together with [12] and [22]).

(3) $PTime = \{K \subseteq Struc(\sigma) \mid \sigma$ is a finite relational structure and K is definable by an $(FO+LFP)(\sigma)$-*sentence*$\}$ [9], [24].

3.2. There are canonical nonuniform counterparts to the classes *LogSpace*, *NLogSpace*, and *PTime*. Among the various mutually equivalent variants used to make precise the notion of *nonuniform computation*, we choose the following approach originating in [14].

$L \subseteq \{0,1\}^*$ is said to be in *NU-LogSpace* if there are a polynomially length bounded advice function $\alpha : \mathbf{N} \to \{0,1\}^*$ and a language $L' \in LogSpace$ such that for all $w \in \{0,1\}^*$, $w \in L$ if and only if $w\hat{}\alpha(|w|) \in L'$. That is, nonuniformity is performed by adding to the input word an advice that is of polynomial length and depends only on the length of the input. The definitions of *NU-NLogSpace* and *NU-PTime* are analogous.

Now we shall show that there are logical characterizations of these classes, analogous to that of the uniform case (see § 3.1), but with FO_{nu} in place of *FO*.

THEOREM 3.3. *Let L be a language over* $\{0,1\}$. *Then*

(1) $L \in$ *NU-LogSpace if and only if L is definable by an* $(FO_{nu}+dTC)(H)$-*sentence.*

(2) $L \in$ *NU-NLogSpace if and only if L is definable by an* $(FO_{nu}+TC)(H)$-*sentence.*

(3) $L \in$ *NU-PTime if and only if L is definable by an* $(FO_{nu}+LFP)(H)$-*sentence.*

The proof is based on the characterization of the corresponding uniform complexity classes (see § 3.1). Since the arguments are similar for each of the three assertions we will point out how to prove (1) only.

Given a finite relational signature σ, it is quite simple to regard finite σ-structures as words over $\{0,1\}$ as follows. Let $\mathcal{A}$ be a σ-structure over the universe $[n]$, and let R be a k-ary relation symbol from σ. Define $code(R^{\mathcal{A}})$ to be the listing of all values of the characteristic function of $R^{\mathcal{A}}$ with respect to the lexicographic order on $[n]^k$. For a constant symbol c from σ, $code(c^{\mathcal{A}})$ is defined analogously. Let $code(\mathcal{A})$ be the concatenation of the codes of $\mathcal{A}$-interpretations of all relation symbols and constant symbols occurring in σ. We will make use of this construct in the proof of (1).

Let us assume first that $L \subseteq \{0,1\}^*$ is in *NU-LogSpace*. Then there are an advice function $\alpha : \mathbf{N} \to \{0,1\}^*$ and a language $L' \in$ *NU-LogSpace* such that

$$L = \{w \in \{0,1\}^* \mid w\hat{}\alpha(|w|) \in L'\}.$$

Without loss of generality we may assume that

(1) $|\alpha(n)| = n^k$ for some $k \in \mathbf{N}$ and

(2) If $u \in L'$, then $|u| = l + l^k$ for some $l \in \mathbf{N}$.

Let σ be a signature consisting of a unary predicate symbol H and a k-ary predicate symbol B, and denote by K' the class of all finite σ-structures A for which $code(\mathcal{A}) \in L'$. Using the notation of [10] and [13], K' is a "problem" belonging to *LogSpace*. From § 3.1 it follows that there is an $(FO+dTC)$-sentence φ' defining K', i.e.,

$$K' = \{\mathcal{A} \in Struc(\sigma) \mid \mathcal{A} \models \varphi'\}.$$

Now replace the occurrences of B in φ' by the *nu*-relation symbol β, where $code(\beta_n) = \alpha(n)$. Denote the resulting $(FO_{nu}+dTC)(H)$-formula by φ. Then for each $w \in \{0,1\}^n$:

$$([n], code(w)) \models \varphi' \Leftrightarrow ([n], code(w), \alpha(n)) \models \varphi,$$

i.e., φ is an $[FO_{nu}+dTC)(H)$-definition of L.

Conversely, assume that a $(FO_{nu}+dTC)(H)$-sentence φ defines L. Replace in φ the *nu*-relation and *nu*-constant symbols by new nonlogical symbols. This gives a sentence φ' in an enriched signature. According to § 3.1, φ' defines a problem K' in *LogSpace*. The code of an arbitrary structure $\mathcal{A} = ([n], H^{\mathcal{A}}, \cdots) \in K'$ consists of

$code(H^{\mathcal{A}})$, followed by the code of the new nonlogical relations and constants. This additional information is of polynomial length and, since it originates in *nu*-relations and constants, it depends only on n. Hence we can use it as an advice function $\alpha: \mathbf{N} \to \{0,1\}^*$. Let $L' = \{code(\mathcal{A}) \mid \mathcal{A} \in K'\}$. Then

$$L = \{w \in \{0,1\} \mid w\hat{}\alpha(n) \in L'\}.$$

But L' is in *LogSpace*, and this completes the proof of (1). □

3.4. Now we turn to complexity classes that are defined by circuits. We shall regard here the nonuniform versions, i.e., a completely separate computational device may be used for each input length n. Recently, these classes have gained more and more interest in complexity theory, and some interesting separation results have been proved [6], [18]-[20].

Let us start with the so-called AC^0-*circuits*. A language $L \subseteq \{0,1\}^*$ is said to be in AC^0 if and only if there is a sequence $(C_n)_{n\in\mathbf{N}}$ of Boolean circuits such that

(1) The circuits consist of *AND*- and *OR*-gates of arbitrary fan-in;

(2) Each C_n has input nodes $I_1, \cdots, I_n, \hat{I}_1, \cdots, \hat{I}_n, 0, 1$ corresponding to n Boolean inputs, their negations and two constant inputs, respectively;

(3) There is a polynomial bound on the size of the C_n's;

(4) The depth of the C_n's is bounded by a constant; and

(5) C_n accepts exactly those inputs belonging to $:= L \cap \{0,1\}^n$.

Admitting additionally MOD_q-gates of arbitrary fan-in, we obtain the classes $AC^0[MOD_q]$ ($MOD_q(w) = 1$ if and only if the sum of the inputs is congruent to 0 modulo q).

AC^0 and its extensions $AC^0[MOD_q]$ ($q \in \mathbf{N}$, $q \geqq 2$) are contained in a wider class called NC^1, which is defined as follows.

$L \subseteq \{0,1\}^*$ belongs to NC^1 if and only if L can be accepted (nonuniformly) by a sequence $(C_n)_{n\in\mathbf{N}}$ of usual combinational circuits such that

(1) Each C_n may consist of binary *AND*-gates, binary *OR*-gates, and unary *NOT*-gates, respectively;

(2) The input nodes of C_n are labeled by $I_1, \cdots, I_n, 0, 1$; and

(3) The depth of C_n is bounded by an $O(\log(n))$.

It is known that AC^0 forms a proper subset of each $AC^0[MOD_q]$ [6], and that, if q is a prime, $AC^0[MOD_q]$ is properly contained in NC^1 (see [18] for $q = 2$ and [20] for the general case). On the other hand, many questions concerning the relations between $AC^0[MOD_q]$ and $AC^0[MOD_r]$ or between $AC^0[MOD_q]$ (q arbitrary) and NC^1 are still open. A structural description of these nonuniform complexity classes has been given in [3], using the framework of nonuniform deterministic finite automata.

3.5. Let us define the logical tools we need to characterize AC^0, $AC^0[MOD_q]$, and NC^1 in terms of expressibility.

Given a logic $L(\sigma)$ and a finite group G, by a *G-representation* in $L(\sigma)$ we understand a set $\hat{G} = \{\hat{g} \mid g \in G\}$ consisting of k-tuples of constant symbols, for some k, such that the mapping $g \mapsto \hat{g}$ is $1-1$.

The expressive power of $L(\sigma)$ can be increased by admitting a new type of formulas which we shall refer to as $(G-\Pi)$-*formulas*. Their syntax is given by the following $(G-\Pi)$-*formation rule*.

Suppose that

(1) $\hat{G}$ is a G-representation (by k-tuples);

(2) For each $g \in G$, $\chi_g(\hat{u})$ is a formula containing the l-tuple $\hat{u}$ of free variables.

Then

$$\hat{z} \in \prod_{\hat{x}}^{\hat{y}} [\chi_g, \hat{G}]$$

is a formula. Here $\hat{x}, \hat{y}$ are l-tuples of variables, and $\hat{z}$ is a k-tuple. The variables $\hat{x}, \hat{y}, \hat{z}$ occur free in the new formula, whereas the variables $\hat{u}$ (from χ_g) are bounded by the $(G-\Pi)$-construct.

To describe the semantics of the new formulas, let $<_l$ denote the usual lexicographic order on $\mathbf{N}^m$. The $(G-\Pi)$-*semantics rule* is

$$\mathscr{A} \models \hat{c} \in \prod_{\hat{a}}^{\hat{b}} [\chi_g, \hat{G}]$$

if and only if

(1) The interpretation of the constant vectors $\hat{g}$ in $\mathscr{A}$ yield pairwise different k-tuples over n;

(2) For any $\hat{d} \in [n]^l$ there is exactly one $g \in G$ such that $\mathscr{A} \models \chi_g(\hat{d})$, i.e., the elements of $[n]^l$ are "colored" by group elements $g \in G$;

(3) Either $\hat{a} = \hat{b}$ and for some $g \in G$,

$$\mathscr{A} \models \chi_g(\hat{a}) \wedge \mathscr{A} \models \hat{g} = \hat{c},$$

or $\hat{a} <_l \hat{b}$ and there are sequences $\hat{a}_0 = \hat{a} <_l \hat{a}_1 <_l \cdots <_l \hat{a}_m = \hat{b}$ of lexicographically consecutive elements of $[n]^l$, $\hat{g}_0, \cdots, \hat{g}_m$ of elements of G, such that

$$\mathscr{A} \models \bigwedge_{i=0,\cdots,m} \chi_{g_i}(\hat{a}_i) \wedge \hat{g} = \hat{c}$$

where

$$g = \prod_{i=0,\cdots,m} g_i \in G.$$

In other words, a $(G-\Pi)$-formula expresses that the G-product of "colors" over an $<_l$-interval is a certain group element. To manage the coloring and to code the value of the G-product, the $(G-\Pi)$-formula needs a G-representation by constants.

Evidently, the $(G-\Pi)$-semantics rule can be expressed by means of the *LFP*-operator. Moreover, in view of the "exactly one" phrase in condition (2), the *dTC*-operator is sufficient.

To illustrate the expressive power of the new formulas, we show how to count modulo q using the $(\mathbf{Z}_q-\Pi)$-construct.

Let $\varphi(x)$ denote an arbitrary $L(\sigma)$-formula, and put $\chi_1(x) \equiv \varphi(x)$, $\chi_0(x) \equiv \neg\varphi(x)$, $\chi_m(x) \equiv \neg(x = x)$ for each $m \in \mathbf{Z}_q$, $m \neq 0, 1$. Assume that $\hat{\mathbf{Z}}_q$ is a $\mathbf{Z}_q$-representation. Then for any $\mathscr{A} \in Struc(\sigma)$,

$$\mathscr{A} \models \hat{0} \in \prod_{1}^{n} [\chi_m, \hat{\mathbf{Z}}_q] \Leftrightarrow card\,\{a \in \mathscr{A} \mid \mathscr{A} \models \varphi(a)\} \equiv 0(q).$$

It is not hard to verify that the $(\mathbf{Z}_q-\Pi)$-construct is of the same expressive power as the sequence of the following quantifiers.

$Q_q^k \hat{x} \varphi(\hat{x}) \Leftrightarrow$ the number of k-tuples satisfying φ is congruent to 0 modulo q.

THEOREM 3.6. *Let L be a language over $\{0, 1\}$. Then*

(1) *$L \in AC^0$ if and only if L is definable by an $FO_{nu}(H)$-sentence.*

(2) *$L \in AC^0[MOD_q]$ if and only if L is definable by an $(FO_{nu}+\mathbf{Z}_q-\Pi)(H)$-sentence*;

(3) $L \in NC^1$ *if and only if* L *is definable by an* $(FO_{nu}+G-\Pi)(H)$*-sentence for some finite nonsolvable group* G.

Part (1) has been proved in [10] and [13] and is a special case of a more general result in [7]. However, it is possible to prove all these results in a unique fashion using the algebraic ideas of [3]. This will be pointed out in § 4.

3.7. The last group of results concerns branching programs. A *branching program* over n Boolean inputs is a finite, directed, acyclic graph with some additional structure:

(1) There are exactly one source node s and two sink nodes a ("accept") and r ("reject").

(2) Each nonsink node is labeled by one of the Boolean input variables $\{x_1, \cdots, x_n\}$ and has outdegree 2.

(3) One of the two edges leaving an arbitrary nonsink node is labeled by 0, the other one is labeled by 1.

Given n input values, there is a unique path from the source to one of the sink nodes: while the actual node is not a sink, check the Boolean input assigned to it and go along the 0- or 1-edge, respectively. The program accepts or rejects the input word depending on whether this path terminates at the a- or r-sink, respectively. The *size* of a branching program is the number of its nonsink nodes.

A language $L \subseteq \{0,1\}^*$ can be computed by a sequence $(P_n)_{n\in\mathbf{N}}$ of polynomial-size branching programs if and only if it belongs to *NU-LogSpace* (see [17] for a proof).

In [15] the notion of a branching program was generalized by introducing a new type of nonsink nodes. Let Ω be a set of binary Boolean functions. In an Ω*-branching program* every nonsink node either is an ordinary query node, as defined above, or it is labeled by an $\omega \in \Omega$. Given an input word w, a Boolean value is assigned to each node of the program by induction:

$$val(a) := 1, \qquad val(r) := 0.$$

If b is a query node labeled by x_i, and if b_0, b_1 are the successors of b via the 0-edge and the 1-edge, respectively, then

$$val(b) := \begin{cases} val(b_0) & \text{if } w_i = 0, \\ val(b_1) & \text{otherwise.} \end{cases}$$

If b is labeled by $\omega \in \Omega$ and has successors as above, then

$$val(b) := \omega(val(b_0), val(b_1)).$$

An Ω-branching program *accepts* $w \in \{0,1\}^*$ if and only if $val(s) = 1$.

It has been pointed out in [15] that there are only four relevant sets Ω of *binary* Boolean functions: $\{\vee\}$, $\{\wedge\}$, $\{MOD_2\}$, and $\{\vee, \wedge\}$.

Denote by $P_{\Omega-BP}$ the class of all languages over $\{0,1\}$ that can be accepted (nonuniformly) by sequences of polynomial size Ω-branching programs. These classes have been characterized [15] as follows.

THEOREM 3.8. (1) $P_{\{\vee\}-BP} = NU - NLogSpace$, $P_{\{\wedge\}-BP} = NU - NLogSpace$.

(2) $NU - LogSpace \subseteq P_{\{MOD_2\}-BP} \subseteq NU - PTime$.

(3) $P_{\{\vee,\wedge\}-BP} = NU - PTime$.

Using the framework of *nu*-logic it is easy to prove the same results. This will be pointed out in § 5.

4. Proof of Theorem 3.6.

Fact 4.1. $FO_{nu}(H)$-definable languages belong to AC^0.

To prove this fact, let φ be an $FO_{nu}(H)$-sentence. For structures with universe $[n]$, we simulate φ by a circuit C_n in such a way that, for all $w \in \{0,1\}^n$,

$$([n], H_w) \models \varphi \Leftrightarrow C_n \text{ accepts } w.$$

We do this by induction on the construction of φ.

If φ is atomic and has the form $H(i)$, $i \in [n]$, then it can be simulated by the ith input node. Atomic sentences that are built from nonuniform relation and constant symbols have fixed evaluations over $[n]$ and hence may be interpreted by constant inputs.

It is evident how to simulate negated atomic formulas.

If $\varphi \equiv \varphi_1 \wedge \varphi_2$, then two circuits simulating φ_1 and φ_2 are connected by an *AND*-gate.

Quantifiers can be simulated by *OR*-gates (*AND*-gates, respectively) of fan-in $= n$: think of $\exists x \varphi(x)$ as $\bigvee_{m \in [n]} \varphi(m)$. Since there is a constant number of quantifier simulations to be executed, the size of the resulting circuit is bounded by a polynomial in n. □

PROPOSITION 4.2. (1) *For an arbitrary finite group G, the $(FO_{nu} + G - \Pi)(H)$-definable languages belong to NC^1.*

(2) *The $(FO_{nu} + \mathbf{Z}_q - \Pi)(H)$-definable languages belong to $AC^0[MOD_q]$.*

Proof. We must introduce some new constructions into the inductive argument of Fact 4.1.

(1) Note that the simulation of $\exists$-quantifiers can be established by binary *OR*-gates in depth $O(\log(n))$. Now suppose that we must simulate $(\hat{a}, \hat{b}, \hat{c}) \in \Pi_{g \in G}[\chi_g, \hat{G}]$ by a circuit, and assume that NC^1-circuits for each of the sentences $\chi_g(\hat{d})$ $(g \in G, \hat{d} \in [n]^k)$ are constructed. Using these circuits, it is easy to check points (1) and (2) of the $(G - \Pi)$-semantics rule (§ 3.5) by combinational circuits of depth $O(\log(n))$. Next we build a combinational circuit of constant depth with $2 \cdot |G|$ inputs and $|G|$ outputs, simulating the multiplication of two elements of G: the output corresponding to $g \in G$ is on if and only if there are $g_1, g_2 \in G$ such that $g_1 \cdot g_2 = g$ and the inputs corresponding to g_1 in the first input block and to g_2 in the second one are on. Using these multiplication circuits and those for $\chi(\hat{d})$, $\hat{d}$ from the $<_k$-interval $[\hat{a}, \hat{b}]$, we can determine the product of "colors" over this interval in depth $O(\log(n))$.

(2) Having $AC^0[MOD_q]$-circuits for $\chi_g(\hat{d})(g \in \mathbf{Z}_q)$, we easily check parts (1) and (2) of the $(G - \Pi)$-semantics rule in constant depth and polynomial size. To handle (3) also, observe that we can check whether the number of elements between $\hat{a}$ and $\hat{b}$ colored by some fixed $g \in \mathbf{Z}_q$ is congruent m modulo q. This is done by feeding $\chi_g(\hat{d})$ for those $\hat{d}$ together with an appropriate number of constant 1 inputs into an MOD_q-gate. Thus we can produce an $AC^0[MOD_q]$-circuit having output nodes for the relations "the number of occurrences of g as a color in $[\hat{a}, \hat{b}]$ is congruent m modulo q." From these outputs we easily construct the desired circuit by adding *AND*- and *OR*-gates, increasing the depth by a constant only. □

4.3. To go the other way around, i.e., to obtain formulas from circuits, we will use the results of [3]. In each of the three cases (Theorem 3.6(1)–(3)) our arguments will be of the same fashion:

(1) To the circuit defined complexity class $\mathcal{K}$ we assign a finite alphabet A.

(2) To every $\{0,1\}$-language $L \in \mathcal{K}$ we assign a word function

$$W_L : \{0,1\}^* \to A^*$$

and an "*acceptance language*" Acc_L over A with the following property.

For all n and for all $w \in \{0,1\}^n$, $w \in L$ if and only if $W_L(w) \in Acc_L$.

(3) We show that the languages Acc_L are definable in an appropriate way. Exploiting some "nice" properties of W_L, we translate the defining sentence for Acc_L into the desired $(FO_{nu}+\cdots)(H)$-sentence.

Parts (1), (2), and the "nice" properties mentioned in (3) are essentially based on the techniques of [3]. We omit details here. Instead, we list the results in a form we need for our proofs.

The alphabets A_0, A_{MOD} corresponding to AC^0 and $AC^0[MOD_q]$ $(q>1)$, respectively, are defined as follows:

$$A_0 = \{0, 1, [_\wedge,]_\wedge, [_\vee,]_\vee\}, \qquad A_{MOD} = \{0, 1, [_\wedge,]_\wedge, [_\vee,]_\vee, [_{MOD},]_{MOD}\}.$$

In the case of NC^1 let us fix a finite nonsolvable group G as our alphabet.

4.4. Let B be a finite alphabet. Remember that a language $L \subseteq B^*$ is called aperiodic if and only if there is a $k \in \mathbf{N}$ such that for all $x, y, z \in B^*$, $xy^k z \in L$ if and only if $xy^{k+1}z \in L$ [16].

To define yet another class of regular languages we need additional notation. Assume that L_0, L_1 are languages over B, and fix some $b \in B$. Then we denote by $[L_0, b, L_1, 0]_q$ the set of all words over B that cannot be written in the form $w_0 b w_1$ with $w_0 \in L_0$, $w_1 \in L_1$. For $0 < k \leqq q$, $[L_0, b, L_1, k]_q$ denotes the set of all words for which the number of factorizations $w_0 b w_1$ $(w_0 \in L_0, w_1 \in L_1)$ is congruent k modulo q. Now we can define a hierarchy of regular languages by induction:

$$M_q^0(B) = \{\phi, B^*\},$$

$$M_q^{i+1}(B) = M_q^i(B) \cup \{[L_0, b, L_1, k]_q \mid b \in B, L_0, L_1 \in M_q^i(B), 0 \leqq k \leqq q\}^{BC} \quad \text{where } BC$$

denotes the closure with respect to Boolean operations.

According to the results of [3] the acceptance languages for AC^0 are regular aperiodic languages over A_0, and in the case of $AC^0[MOD_q]$ they belong to $\bigcup_{i \in \mathbf{N}} M_q^i(A_{MOD})$. Moreover, to an $L \in NC^1$ we can assign a word function W_L; $\{0,1\}^* \to G^*$ such that $w \in L$ if and only if $\prod W_L(w) = e$, where $\prod$ denotes the product taken in G and e denotes the neutral element of G.

This completes the list of results related to (1) and (2) of § 4.3.

FACT 4.5. Let W_L be one of the word functions from (2) of § 4.3, and let A denote the corresponding alphabet. Then:

(1) If $|w_1| = |w_2|$, then $|W_L(w_1)| = |W_L(w_2)|$;

(2) The length $|W_L(w)|$ on inputs w of length n is polynomially bounded;

(3) For fixed $n \in \mathbf{N}$, the letter on a fixed position m in $W_L(w)$ $(w \in \{0,1\}^n)$ depends on at most one input. More precisely, either all $W_L(w)$ $(w \in \{0,1\}^n)$ have the same letter on position m, or there are two letters $b_0, b_1 \in A$ and some $i \in \mathbf{N}$ such that $w_i = 0 \Rightarrow$ in $W_L(w)$, b_0 is on position m, $w_i = 1 \Rightarrow$ in $W_L(w)$, b_1 is on position m.

These are the "nice" properties cited in (3) of § 4.3.

4.6. In order to get a link from these results to logic we sketch how to regard words over an alphabet B as structures of a logic $FO_W(\sigma_B)$. We will call those structures *word models*, and later we will use them to produce the desired logical descriptions of AC^0, $AC^0[MOD_q]$, and NC^1.

Fix an arbitrary finite alphabet B. As symbols of FO_W we take those of FO. The signature σ_B contains, for each $b \in B$, a unary predicate symbol P^b. The syntax is as usual. Structures have as universes one of the sets $[n]$ $(n \in \mathbf{N})$ and satisfy the following restriction on the interpretation of the predicate symbols of σ_B.

For each $m \in [n]$ there is exactly one $b \in B$ (the letter at position m) such that m satisfies P^b.

Thus we can think of an $FO_W(\sigma_B)$-structure over $[n]$ as a B-word of length n and vice versa.

Of course it is possible to add *LFP*-constructs to FO_W. In order to make sure that the $(G-\Pi)$-construct (see § 3.5) always gives a sense, we include a G-representation into the logical symbols, thus obtaining $(FO_{W,\hat{G}}+G-\Pi)(\sigma_B)$. According to the convention that all structures under consideration are of cardinality greater than or equal to two, two new constant symbols will be enough to construct a G-representation.

LEMMA 4.7. (1) *L is a regular aperiodic language over B if and only if L is definable by a $FO_W(\sigma_B)$-sentence*;

(2) *$L\in\bigcup_{i\in\mathbf{N}} M_q^i(B)$ if and only if L is definable by an $(FO_{W,\hat{\mathbf{Z}}_q}+\mathbf{Z}_q-\Pi)(\sigma_B)$-sentence*;

(3) *If G is a finite group, then the $(FO_{W,\hat{G}}+G-\Pi)(\sigma_G)$-sentence*

$$\hat{e}\notin \prod_{\min}^{\max}[P^g(x),\hat{G}]$$

expresses that the product of the letters of a G-word differs from $e\in G$. (Here $\hat{e}$ denotes the sequence of constant symbols representing $e\in G$ as regards $\hat{G}$.)

Part (1) is a well-known theorem (see, for example, [16] or [23]). The proof of (2) is based on the fact that the $(\mathbf{Z}_q-\Pi)$-construct admits counting modulo q (see the remark at the end of § 3.5). The crucial step in the proof is to construct sentences defining languages of the type $[L_0, b, L_1, k]_q$. Part (3) is obvious. □

4.8. Now we are in position to prove the converse of Fact 4.1.

Fix an arbitrary language $L\in AC^0$ and apply Fact 4.5 of (2) to find a constant $k\in\mathbf{N}$ such that the words $W_L(w)$ $(w\in\{0,1\}^n)$ (from (2) of § 4.3) are of length at most n^k. So we can identify the positions in the words $W_L(w)$ $(w\in\{0,1\}^n)$ with an initial segment of the k-tuples over $[n]$ in their lexicographic order.

Now we apply (1) and (3) of § 4.3 to define, for all letters $a, a_0, a_1\in A_0$, the following nonuniform relations:

$P_n^a(r_1,\cdots,r_k)\Leftrightarrow$ all words $W_L(w)$ $(w\in\{0,1\}^n)$ have the letter a at the position coded by $(r_1,\cdots,r_k)$.

$P_n^{\alpha_0{}^{a_1}}(i,r_1,\cdots,r_k)\Leftrightarrow$ for all words $W_L(w)$ $(w\in\{0,1\}^n)$, if $w_i=0$ then a_0 is at position $(r_1,\cdots,r_k)$, otherwise a_1 is at the position coded by $(r_1,\cdots,r_k)$.

Furthermore, we define $\widehat{\min}$ and $\widehat{\max}$ to be k-tuples of nonuniform constants coding the beginning and the end of the words $W_L(w)$ $(w\in\{0,1\}^n)$, respectively.

Using the *nu*-symbols for these new relations and constants, we show how an arbitrary $FO_W(\sigma_{A_0})$-sentence φ can be translated into an $FO_{nu}(H)$-sentence φ' such that

$$W_L(w)\models_{FO_W(\sigma_{A_0})}\varphi\Leftrightarrow([n],H_w)\models_{nu}\varphi'.$$

Variables in φ become k-tuples of variables in φ', we do the same with the constant symbols min and max. The relation symbols $<$ and $=$ are translated into $<_k$ and $=_k$, the *nu*-symbols for lexicographic order and identity of k-tuples, respectively.

Any atomic formula of the type $P^a(x)$ $(a\in A_0)$ will be transformed to

$$\mathbf{P}^a(x_1,\cdots,x_k)$$
$$\vee\exists i\left(\neg H(i)\wedge\bigvee_{b\in A_0}\mathbf{P}^{a,b}(i,x_1,\cdots,x_k)\right)\vee\exists i\left(H(i)\wedge\bigvee_{b\in A_0}\mathbf{P}^{b,a}(i,x_1,\cdots,x_k)\right).$$

Finally, the $\exists$-quantifiers of $FO_W(\sigma_{a_0})$ of the form $\exists x\cdots$ are translated into

$$\exists x_1\cdots\exists x_k(\min\leqq_k(x_1,\cdots,x_k)\leqq_k\max\wedge\cdots).$$

According to Lemma 4.7 there is an $FO_W(\sigma_{A_0})$-sentence φ_L defining the aperiodic language Acc_L (see § 4.3(2)). To φ_L we apply the translation procedure just described, thus obtaining the desired $FO_{nu}(H)$-sentence defining $L \in AC^0$. This completes the proof of Theorem 3.6(1).

Similar arguments are applied in the proofs of Theorem 3.6(2) and (3). □

5. Proof of Theorem 3.8.

5.1. Assume $(P_n)_{n\in\mathbf{N}}$ to be a sequence of polynomial-size Ω-branching programs. Without loss of generality we may assume that $size(P_n) \leqq n^k$ for some $k \in \mathbf{N}$. Hence it is possible to enumerate the nodes of each P_n by elements of $[n]^k$. Having fixed such enumerations, nonuniform relations can be defined as follows:

$Query_n(x_1, \cdots, x_k, i, y_1, \cdots, y_k, z_1, \cdots, z_k) \Leftrightarrow$ the k-tuple $\hat{x}$ numbers a query node in P_n, labeled by the ith input, and the corresponding 0- and 1-edges lead to the nodes of P_n with the numbers $\hat{y}$ and $\hat{z}$, respectively.

$\omega - Branch_n(x_1, \cdots, x_k, y_1, \cdots, y_k, z_1, \cdots, z_k) \Leftrightarrow \hat{x}$ numbers a nonsink node in P_n, labeled by the $\omega \in \Omega$, and the corresponding 0- and 1-edges lead to the nodes of P_n with the numbers $\hat{y}$ and $\hat{z}$, respectively.

Furthermore, select nonuniform constants

$$(s_n^i)_{n\in\mathbf{N}}, (a_n^i)_{n\in\mathbf{N}}, (r_n^i)_{n\in\mathbf{N}}, \qquad (1 \leqq i \leqq k),$$

such that $(s_n^1, \cdots, s_n^k)$, $(a_n^1, \cdots, a_n^k)$, $(r_n^1, \cdots, r_n^k)$ are the numbers of the source, accepting and rejecting nodes in the enumeration of P_n, respectively.

Now we are in position to invoke the result Theorem 3.3.

5.2. If $\Omega = \{\bigvee\}$ then the language accepted by $(P_n)_{n\in\mathbf{N}}$ is defined by the following $(FO_{nu} + TC)(H)$-sentence:

$$\begin{aligned}(\hat{\mathbf{s}}, \hat{\mathbf{a}}) \in TC_{\hat{u}\hat{x}}[\hat{u} =_k \hat{v} \vee \exists i \exists \hat{w}(&\mathbf{Query}(\hat{u}, i, \hat{v}, \hat{w}) \wedge \neg H(i)) \\ \vee \exists i \exists \hat{w}(&\mathbf{Query}(\hat{u}, i, \hat{w}, \hat{v}) \wedge H(i)) \\ \vee \exists \hat{w}(\bigvee\text{-}&\mathbf{Branch}(\hat{u}, \hat{v}, \hat{w})) \vee \exists \hat{w}(\bigvee\text{-}\mathbf{Branch}(\hat{u}, \hat{w}, \hat{v}))].\end{aligned}$$

In the case of $\Omega = \{\bigwedge\}$ an analogous formula would work.

If $\Omega = \{\bigwedge, \bigvee\}$, then the following $(FO_{nu} + LFP)(H)$-sentence will do the job:

$$\begin{aligned}\hat{\mathbf{s}} \in LFP_{\hat{u},P}[\hat{u} =_k \hat{\mathbf{a}} \vee \exists i \exists \hat{w}_0 \exists \hat{w}_i(&\mathbf{Query}(\hat{u}, i, \hat{w}_0, \hat{w}_1) \wedge \neg H(i) \wedge P(\hat{w}_0)) \\ \vee \exists i \exists \hat{w}_0 \exists \hat{w}_i(&\mathbf{Query}(\hat{u}, i, \hat{w}_0, \hat{w}_1) \wedge H(i) \wedge P(\hat{w}_1)) \\ \vee \exists \hat{w}_0 \exists \hat{w}_1(\bigwedge\text{-}&\mathbf{Branch}(\hat{u}, \hat{w}_0, \hat{w}_1) \wedge P(\hat{w}_0) \wedge P(\hat{w}_1)) \\ \vee \exists \hat{w}_0 \exists \hat{w}_1(\bigvee\text{-}&\mathbf{Branch}(\hat{u}, \hat{w}_0, \hat{w}_1) \wedge P(\hat{w}_0)) \vee \exists \hat{w}_0 \exists \hat{w}_1(\bigvee\text{-}\mathbf{Branch} \\ &(\hat{u}, \hat{w}_0, \hat{w}_1) \wedge P(\hat{w}_1))].\end{aligned}$$

In the case of $\Omega = \{MOD_2\}$ we must be more careful to keep the "induction predicate symbol" P positive in the formula. This can be established by representing the possible values 0 and 1 of nodes in P_n (see the definition of acceptance behavior in § 3.7) by an additional variable. Set

$$\begin{aligned}\psi(\hat{u}, v) \equiv [(v = \min \wedge \hat{u} =_k \hat{\mathbf{r}}) \vee (v = \max \wedge \hat{u} =_k \hat{\mathbf{a}})] \\ \vee \exists i \exists \hat{w}_0 \exists \hat{w}_1(\mathbf{Query}(\hat{u}, i, \hat{w}_0, \hat{w}_1) \wedge \neg H(i) \wedge P(\hat{w}_0, v)) \\ \vee \exists i \exists \hat{w}_0 \exists \hat{w}_1(\mathbf{Query}(\hat{u}, i, \hat{w}_0, \hat{w}_1) \wedge H(i) \wedge P(\hat{w}_1, v)) \\ \vee \exists \hat{w}_0 \exists \hat{w}_1 \exists \hat{v}_0 \exists \hat{v}_1(\mathbf{MOD}_2 - \mathbf{Branch}(\hat{u}, \hat{w}_0, \hat{w}_1) \wedge P(\hat{w}_0, v_0) \\ \wedge P(\hat{w}_1, v_1) \wedge v_0 = v_1)],\end{aligned}$$

then

$$(\hat{t}, \max) \in LFP_{\hat{u},v,P}[\psi(\hat{u}, v)]$$

expresses that the node of P_n with number $\hat{t}$ has value 1 with regard to the input word that is coded by H. Hence

$$(\hat{s}, \max) \in LFP_{\hat{u},v,P}[\psi(\hat{u}, v)]$$

is the desired sentence.

5.3. To complete the proof of Theorem 3.8 it remains to verify

$$NU - NLogSpace \subseteq P_{(\vee)-BP} \quad \text{and} \quad NU - PTime \subseteq P_{(\vee,\wedge)-BP}.$$

We will do this by using other characterizations of the nonuniform complexity classes *NU-NLogSpace* and *NU-PTime*, respectively. For more background we refer the reader to [17]. A language $L \subseteq \{0, 1\}^*$ belongs to *NU-NLogSpace* if and only if it can be recognized by a sequence of polynomial-size directed switching networks. These networks can be simulated by $\{\vee\}$-branching programs of size at most quadratic in the network size: branchings in the network, i.e., vertices of outdegree $\geqq 2$, become iterated $\{\vee\}$-branchings, whereas the switching edges correspond to usual query nodes. This proves (1) of Theorem 3.8. Part (2) is a consequence of (1) and the results of [12] and [22].

To conclude with 3.8.3 we remark that a language $L \subseteq \{0, 1\}^*$ is in *NU-PTime* if and only if it can be accepted by a sequence of polynomial-size combinational circuits over $\{\vee, \wedge\}$ (fan-in 2!) and Boolean inputs $I_1, \cdots, I_n, \hat{I}_1, \cdots, \hat{I}_n, 0, 1$. But these circuits trivially may be regarded as $\{\vee, \wedge\}$-branching programs: simply replace the inputs and negated inputs by corresponding query nodes leading to a or r, respectively. □

6. Concluding remarks. We have tried to show that several nonuniform complexity classes can be characterized by the expressive tools that are needed to formulate a computational problem belonging to these classes. From the separation results in [18] and [20] it follows that some of the logical arguments we used essentially differ in their expressive power. Meanwhile, similar approaches to nonuniform complexity classes have been discovered independently by some other researchers, also [1], [2], [21]. An alternative characterization of NC^1 has been given in [4]. However, the question remains open whether new separations can be proved using logical characterizations of the complexity classes in question.

Acknowledgments. I thank B. Graw, C. Meinel, and S. Waack, who have supported this research by many discussions and encouraged me to write this paper. Thanks also to a referee who gave useful comments on this paper.

REFERENCES

[1] D. A. MIX BARRINGTON, K. COMPTON, H. STRAUBING, AND D. THÉRIEN, *Regular languages in NC^1*, Tech. Report BCCS-88-02, Boston College, Boston, MA, 1988.

[2] D. A. MIX BARRINGTON, N. IMMERMAN, AND H. STRAUBING, *On uniformity within NC^1*, COINS Tech. Report 88-60, University of Massachusetts, Boston, MA, 1988.

[3] D. A. BARRINGTON AND D. THÉRIEN, *Finite monoids and the fine structure of NC^1*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 101–109.

[4] K. J. COMPTON AND C. LAFLAMME, *An algebra and a logic for NC^1*, in Proc. Logic in Computer Science '88, IEEE Press, 1988, pp. 12–21.

[5] R. FAGIN, *Generalized first-order spectra and polynomial-time recognizable sets*, in Complexity of Computation, R. Karp, ed., SIAM-AMS Proc. 7, American Mathematical Society, Providence, RI, 1974, pp. 27–41.

[6] M. FURST, J. B. SAXE, AND M. SIPSER, *Parity, circuits, and the polynomial-time hierarchy*, in Proc. 22nd Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1981, pp. 260–270.

[7] Y. GUREVICH AND H. R. LEWIS, *A logic for constant depth circuits*, Inform. and Control, 61 (1984), pp. 65–74.

[8] Y. GUREVICH, *Toward logic tailored for computational complexity*, in Computation and Proof Theory, Proc. Logic Coll., Aachen, 1983, M. M. Richter et al., eds., pp. 175–216; Lecture Notes in Mathematics 1104, 1984.

[9] N. IMMERMAN, *Relational queries computable in polynomial time*, in Proc. 14th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1982, pp. 76–98.

[10] ———, *Languages which capture complexity classes*, in Proc. 15th Annual Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 347–354.

[11] ———, *Expressibility as a complexity measure: Results and directions*, Department of Computer Science TR 538, Yale University, New Haven, CT, April, 1987; see also Proc. 2nd Structure in Complexity Theory Conference, 1987, pp. 194–202.

[12] ———, *Nondeterministic space is closed under complement*, Department of Computer Science TR 552, Yale University, New Haven, CT, July 1987.

[13] ———, *Languages that capture complexity classes*, SIAM J. Comput., 16 (1987), pp. 761–778.

[14] M. A. KARP AND R. J. LIPTON, *Some connections between nonuniform and uniform complexity classes*, in Proc. 12th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1980, pp. 302–309.

[15] C. MEINEL, *The power of polynomial-size Ω-branching programs*, in Proc. STACS 88, Bordeaux, R. Cori and M. Wirsing, eds., 1988, pp. 81–90; Lecture Notes in Computer Science, 294 (1988).

[16] R. MCNAUGHTON AND S. PAPERT, *Counter Free Automata*, MIT Press, Cambridge, MA, 1971.

[17] P. PUDLÁK, *The hierarchy of Boolean circuits*, Preprint Nr. 20, CSAV, 1986; see also Computers and Artificial Intelligence, 6 (1987), pp. 449–468.

[18] A. A. RAZBOROW, *Lower bounds on the size of bounded-depth networks over the basis* $\{\wedge, \oplus\}$, Moscow State University, Moscow, USSR, 1986. (In Russian.)

[19] ———, *Lower bounds on the size of bounded-depth networks over a complete basis containing the logical addition*, Mat. Zametki, 41 (1987), pp. 598–607. (In Russian.)

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

[21] H. STRAUBING, D. THÉRIEN, AND W. THOMAS, *Regular languages defined with generalized quantifiers*, in Proc. 15th ICALP, 1988, pp. 561–575.

[22] R. SZELEPCZÉNYI, *The method of forcing for nondeterministic automata*, EATCS Bull., 33 (1987), pp. 96–99.

[23] W. THOMAS, *Classifying regular events in symbolic logic*, J. Comput. System Sci., 25 (1982), pp. 360–376.

[24] M. VARDI, *Complexity of relational query languages*, in Proc. 14th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1982, pp. 302–309.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 424-437, June 1990

DECIDING EQUIVALENCE OF FINITE TREE AUTOMATA*

HELMUT SEIDL†

Abstract. It is shown that for every constant m it can be decided in polynomial time whether or not two m-ambiguous finite tree automata are equivalent. In general, inequivalence for finite tree automata is DEXPTIME-complete with respect to logspace reductions, and PSPACE-complete with respect to logspace reductions, if the automata in question are supposed to accept only finite languages. For finite tree automata with weights in a field R, a polynomial time algorithm is presented for deciding ambiguity-equivalence, provided R-operations and R-tests for 0 can be performed in constant time. This result is used to construct an algorithm deciding ambiguity-inequivalence of finite tree automata in randomized polynomial time. Finally, for every constant m it is shown that it can be decided in polynomial time whether or not a given finite tree automaton is m-ambiguous.

Key words. finite tree automata, equivalence, complexity, ambiguity, semirings

AMS(MOS) subject classifications. 68D35, 68D30, 68F10, 68C25, 20M35

0. Introduction. Finite tree automata were defined in the late sixties by Thatcher and Wright and Doner as generalizations of finite automata accepting word languages to finite state devices for tree languages [ThaWri68], [Do70]. Their main interest in tree automata was a logical point of view. Finite tree automata can describe classes of (finite or infinite) models for formulas of monadic theories with multiple successors and therefore can be used to get effective decision procedures for these theories [ThaWri68], [Do70], [Ra69], [Tho84]. For other possible applications see [GeStei84].

In this paper we investigate finite tree automata accepting tree languages of finite trees from a complexity theoretical point of view. Especially, we want to analyze the equivalence problem for finite tree automata. Two finite tree automata A_1 and A_2 are called equivalent, if and only if they accept the same tree language. We find that the inequivalence problem for finite tree automata is logspace complete in DEXPTIME, and the inequivalence problem for finite tree automata that accept only finite languages is still logspace complete in PSPACE. These problems for finite word automata are known to be PSPACE- and NP-complete with respect to logspace reductions, respectively. Actually, our proofs extend proofs of the corresponding results for finite word automata.

Since the equivalence problem for finite tree automata is provably difficult in general, it seems natural to look for adequate subclasses such that equivalence can be decided in polynomial time, at least for automata of such a restricted class. One parameter that attracts attention in this context is the degree of ambiguity. A finite automaton is called m-ambiguous, if for every input there are at most m accepting computations. In [SteHu81], [SteHu85] Stearns and Hunt III give a polynomial time algorithm that decides equivalence of m-ambiguous finite word automata for every fixed constant m. They employ difference equations for their decision procedure. Kuich [Kui88] simplifies their constructions. Although not stated explicitly, Kuich's proof can be used to show that the equivalence problem for m-ambiguous finite word automata is even in NC. Kuich uses semiring automata and the formalism of formal power series. Semiring automata are obtained from ordinary automata by giving weights

* Received by the editors April 25, 1988; accepted for publication (in revised form) May 22, 1989. A previous version of my paper appeared in the proceedings of STACS'89 (Springer, Lecture Notes in Computer Science 349, pp. 480-492).

† Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Federal Republic of Germany.

in some semiring to transitions and initial states. In particular, Kuich shows how the equivalence problem for m-ambiguous word automata can be reduced to the ambiguity-equivalence problem for unambiguous automata with weights in the field of rational numbers $\mathbb{Q}$.

For tree languages the concept of formal power series seems to be much more involved than for word languages [BeReu82]. Therefore, we avoid using this concept, but show how some basic ideas of Kuich can be carried over to tree automata. Thus, we introduce finite tree automata with weights in some semiring R and show how to reduce the equivalence problem for m-ambiguous finite tree automata to the ambiguity-equivalence problem for unambiguous finite tree automata with weights in $\mathbb{Q}$ or even in $\mathbb{Z}_p$ for some prime number $p > m$.

We then consider the problem of deciding ambiguity-equivalence for finite tree automata with weights in a field R. We are able to prove the appropriate generalization of Eilenberg's equality theorem [Ei74, Thm. 8.1] to tree automata, i.e., we give an explicit upper bound for the depth of a witness for ambiguity-inequivalence. Furthermore, analyzing the proof we get a polynomial time algorithm that decides ambiguity-equivalence of finite tree automata with weights in R, provided we are allowed to perform R-operations and R-tests for 0 in constant time. Note that this does not automatically lead to a polynomial time algorithm deciding ambiguity-equivalence for ordinary finite tree automata (viewed as automata with weights in $\mathbb{Q}$), since the only upper bound for the lengths of occurring integers given by the algorithm is exponential in the input size. However, we get a polynomial time algorithm deciding equivalence of m-ambiguous tree automata. As another consequence, we are able to construct a randomized polynomial algorithm deciding ambiguity-inequivalence of arbitrary finite tree automata.

For reasons of completeness we finally show that it is also possible to detect whether or not our polynomial equivalence test is applicable, i.e., it can be decided in polynomial time for every constant m whether or not a given finite tree automaton has a degree of ambiguity less than m.

A subsequent paper [Se89] considers the finite degree of ambiguity for its own sake, showing that it can be decided in polynomial time whether or not the degree of ambiguity of a finite tree automaton is finite. Furthermore, we will give a tight upper bound for the maximal degree of ambiguity of a finitely ambiguous finite tree automaton.

1. General notations and concepts. In this section we give basic definitions and state some fundamental properties. Especially, we show that for the equivalence or ambiguity-equivalence problems of finite tree automata, it suffices to consider finite tree automata of rank $\leqq 2$.

A ranked alphabet Σ is the disjoint union of alphabets $\Sigma_0, \cdots, \Sigma_L$. For $a \in \Sigma$, the rank of a, $rk(a)$, equals m if and only if $a \in \Sigma_m$. T_Σ denotes the free Σ-algebra of (finite ordered Σ-labeled) trees, i.e., T_Σ is the smallest set T satisfying (i) $\Sigma_0 \subseteq T$, and (ii) if $a \in \Sigma_m$ and $t_0, \cdots, t_{m-1} \in T$, then $a(t_0, \cdots, t_{m-1}) \in T$. Note that (i) can be viewed as the subcase of (ii) with $m = 0$.

The depth of a tree t, $depth(t)$, is defined by $depth(t) = 0$ if $t \in \Sigma_0$, and $depth(t) = 1 + \max\{depth(t_0), \cdots, depth(t_{m-1})\}$ if $t = a(t_0, \cdots, t_{m-1})$ for some $a \in \Sigma_m$, $m > 0$. Let $\mathbb{N}_0$ denote the set of all nonnegative integers, and let $\mathbb{N}_0^*$ denote the set of all finite sequences of nonnegative integers. The set of nodes of t, $S(t)$, is the subset of $\mathbb{N}_0^*$ defined by

$$S(t) = \{\varepsilon\} \cup \bigcup_{j=0}^{m-1} j \cdot S(t_j)$$

where $t = a(t_0, \cdots, t_{m-1})$ for some $a \in \Sigma_m$ with $m \geqq 0$. t defines a map $\lambda_t(\_): S(t) \to \Sigma$, mapping the nodes of t to their labels. We have

$$\lambda_t(r) = \begin{cases} a & \text{if } r = \varepsilon \\ \lambda_{t_j}(r') & \text{if } r = j \cdot r' \end{cases}.$$

A finite tree automaton (abbreviated: FTA) is a quadruple $A = (Q, \Sigma, Q_I, \delta)$, where
Q is a finite set of states,
$Q_I \subseteq Q$ is the set of initial states,
Σ is a ranked alphabet, and
$\delta \subseteq \bigcup_{m \geqq 0} Q \times \Sigma_m \times Q^m$ is the set of transitions of A.
$rk(A) = \max\{rk(a) \mid a \in \Sigma\}$ is called the rank of A.

Let $t = a(t_0, \cdots, t_{m-1}) \in T_\Sigma$ and $q \in Q$. A q-computation of A for t consists of a transition $(q, a, q_0, \cdots, q_{m-1}) \in \delta$ for the root and q_j-computations of A for the subtrees t_j, $j \in \{0, \cdots, m-1\}$. Especially, for $m = 0$, there is a q-computation of A for t if and only if $(q, a, \varepsilon) \in \delta$. Formally, a q-computation ϕ of A for t can be viewed as a map $\phi: S(t) \to Q$ satisfying (i) $\phi(\varepsilon) = q$ and (ii) if $\lambda_t(r) = a \in \Sigma_m$, then $(\phi(r), a, \phi(r \cdot 0) \cdots \phi(r \cdot (m-1))) \in \delta$. ϕ is called accepting computation of A for t, if ϕ is a q-computation of A for t with $q \in Q_I$. For $t \in T_\Sigma$ and $q \in Q$, $\Phi_{A,q}(t)$ denotes the set of all q-computations of A for t, and $\Phi_{A,Q_I}(t)$ denotes the set of all accepting computations of A for t.

$n_A(t)_q = \#\Phi_{A,q}(t)$ is the number of different q-computations of A for t, the Q-tuple $(n_A(t)_q)_{q \in Q}$ is denoted by $n_A(t)$; finally $da_A(t) = \#\Phi_{A,Q_I}(t)$, the number of different accepting computations of A for t, is called the ambiguity of A for t.

The following proposition is an easy consequence of these definitions.

PROPOSITION 1.1. *Assume* $t = a(t_0, \cdots, t_{m-1}) \in T_\Sigma$. *Then*

$$(1) \quad n_A(t)_q = \sum_{(q, a, q_0 \cdots q_{m-1}) \in \delta} n_A(t_0)_{q_0} \cdot \cdots \cdot n_A(t_{m-1})_{q_{m-1}}$$

$$(2) \quad da_A(t) = \sum_{q \in Q_I} n_A(t)_q. \qquad \square$$

The (tree) language accepted by A, $L(A)$, is defined by $L(A) = \{t \in T_\Sigma \mid \Phi_{A,Q_I}(t) \neq \emptyset\}$. The degree of ambiguity of A, $da(A)$, is defined by $da(A) = \sup\{da_A(t) \mid t \in T_\Sigma\}$. A is called
-unambiguous, if $da(A) \leqq 1$;
-ambiguous, if $da(A) > 1$;
-m-ambiguous, if $da(A) \leqq m$;
-finitely ambiguous, if $da(A) < \infty$; and
-infinitely ambiguous, if $da(A) = \infty$.

Two FTAs A_1, A_2 are called
(1) ambiguity-equivalent (written $A_1 \equiv A_2$), iff $da_{A_1}(t) = da_{A_2}(t)$ for all $t \in T_\Sigma$.
(2) equivalent, iff $L(A_1) = L(A_2)$.

Clearly, $A_1 \equiv A_2$ implies $L(A_1) = L(A_2)$. Moreover, if A_1 and A_2 are unambiguous, then $A_1 \equiv A_2$ if and only if $L(A_1) = L(A_2)$.

For describing our algorithms we will mostly use Random Access Machines (RAMs) with the uniform cost criterion (see [Aho74] or [Paul78] for precise definitions and basic properties). If we allow multiplications, divisions, or the manipulation of

registers not containing nonnegative integers, we will state this explicitly. For measuring the computational costs of our algorithms relative to the size of the input automata in question, we define

$$|A| = \sum_{(q,a,q_0\cdots q_{m-1})\in\delta} (m+2).$$

An FTA $A=(Q, \Sigma, Q_I, \delta)$ is called reduced, if

$-\forall m \geqq 0,\ a \in \Sigma_m$: $Q \times \{a\} \times Q^m \cap \delta \neq \varnothing$.

$-\forall q \in Q\ \ \exists t \in T_\Sigma,\ \phi \in \Phi_{A,Q_I}(t)$: $q \in \mathrm{im}(\phi)$.[1]

The following proposition is well known:

PROPOSITION 1.2. *For every* FTA $A=(Q, \Sigma, Q_I, \delta)$ *there is an* FTA $A_r = (Q_r, \Sigma_r, Q_{r,I}, \delta_r)$ *with the following properties*:

(1) $Q_r \subseteq Q$, $Q_{r,I} \subseteq Q_I$, $\delta_r \subseteq \delta$;

(2) A_r *is reduced*; *and*

(3) $A_r \equiv A$.

A_r *can be constructed from* A *by a* RAM (*without multiplications*) *in time* $O(|A|)$. □

Actually, the construction of A_r is analogous to the reduction of a context-free grammar.

Next, we observe that we may restrict our attention without loss of generality to automata of rank $\leqq 2$ (nonetheless we always will give the constructions for arbitrary rank L).

Consider the (injective) tree homomorphism θ that maps symbols of rank >2 onto a tree of binary symbols as shown in Fig. 1. One easily proves:

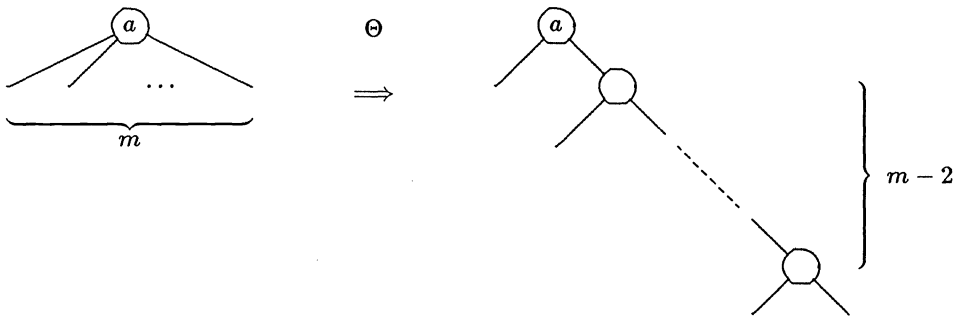


FIG. 1

PROPOSITION 1.3. *For every* FTA $A=(Q, \Sigma, Q_I, \delta)$ *exists an* FTA A' *with the following properties*:

(1) $da_A(t) = da_{A'}(\theta(t))$ *for all* $t \in T_\Sigma$;

(2) $L(A') = \theta(L(A))$.

(3) A' *can be computed from* A *in time* $O(|A|)$. □

2. The complexity of the inequivalence problem. Before we give a polynomial time algorithm that decides equivalence of m-ambiguous FTAs, we analyze the complexity of the inequivalence problems for unrestricted FTAs and FTAs accepting finite languages, respectively. For nondeterministic finite word automata these two problems are known to be PSPACE-complete and NP-complete (with respect to logspace reductions), respectively [MeSto72], [StoMe73]. We find these problems for FTAs to be DEXPTIME-complete and PSPACE-complete, respectively.

[1] $\mathrm{im}(\phi)$ denotes the image of the map ϕ.

For the upper bounds we construct Turing machines that simulate the computations of the subset automata corresponding to the FTAs to be tested for inequivalence. For an FTA $A=(Q, \Sigma, Q_I, \delta)$ the corresponding subset automaton $P(A)$ is defined by

$P(A)=(P(Q), \Sigma, \bar{Q}_I, \bar{\delta})$, where

$P(Q)$ denotes the power set of Q,

$\bar{Q}_I=\{Q' \subseteq Q \mid Q' \cap Q_I \neq \emptyset\}$, and

$(Q', a, Q'_0 \cdots Q'_{m-1}) \in \bar{\delta}$ iff $Q'=\{q \in Q \mid \exists q_0 \in Q'_0, \cdots, q_{m-1} \in Q'_{m-1}: (q, a, q_0 \cdots q_{m-1}) \in \delta\}$.

Let $t \in T_\Sigma$. Define $Q(t)=\{q \in Q \mid \Phi_{A,q}(t) \neq \emptyset\}$. Then $Q(t)$ is the unique subset Q' of Q such that there is a Q'-computation of $P(A)$ for t. This uniqueness is usually called "bottom-up determinism." It is well known that $L(P(A))=L(A)$ [Do70]. Especially, $t \notin L(A)$ if and only if $Q(t) \cap Q_I=\emptyset$.

THEOREM 2.1. (1) *For two* FTAs A_1, A_2 *it can be decided in time* $O(2^{c \cdot n \cdot L})$ *whether or not* $L(A_1) \neq L(A_2)$, *where* L *is the maximal rank,* n *is the maximal number of states of* A_1 *and* A_2, *and* $c>0$ *is a suitable constant independent of* n *and* L.

(2) *The inequivalence problem for* FTAs *is hard in* DEXPTIME *with respect to* logspace *reductions.*

Proof. The proof is an appropriate generalization of a proof showing PSPACE-completeness for deciding inequivalence of finite word automata. Where in the word case, the computations of the subset automata could be simulated by a nondeterministic polynomially space bounded Turing machine, we need an alternating Turing machine in the tree case. For the hardness part instead of coding the word problem for nondeterministic linear space bounded Turing machines into the inequivalence problem, we can, in the tree case, encode the word problem for linear space bounded alternating Turing machines into the inequivalence problem.

Ad(1): Construct an alternating Turing machine M that on input (A_1, A_2) simulates topdown the computation of $P(A_1)$ and $P(A_2)$ for a witness for inequivalence. M needs space $O(L \cdot n)$ on its worktape to verify that for the current pair of state sets $(\bar{Q}^{(1)}, \bar{Q}^{(2)})$, $\bar{Q}^{(i)} \subseteq Q_i$, $i=1,2$, guessed $a \in \Sigma$ and guessed tuple of pairs $(\bar{Q}^{(1)}_0, \bar{Q}^{(2)}_0) \cdots (\bar{Q}^{(1)}_{m-1}, \bar{Q}^{(2)}_{m-1})$, $(\bar{Q}^{(i)}, a, \bar{Q}^{(i)}_0 \cdots \bar{Q}^{(i)}_{m-1})$ is a transition of $P(A_i)$, $i=1,2$.

Since by [ChaKoSto81] $\mathrm{ASPACE}(L \cdot n)=\bigcup_{c>0} \mathrm{DTIME}(2^{c \cdot L \cdot n})$, assertion (1) follows.

Ad(2): Assume M is an alternating n-space bounded Turing machine with tape alphabet Γ, and $w \in \Gamma^N$ for a fixed natural number N. We construct a ranked alphabet Σ and an FTA A with alphabet Σ having $O(N)$ states such that $L(A) \neq T_\Sigma$ if and only if M has an accepting computation for w. This construction will be possible in logspace. Thus, every problem in ASPACE (n) can be reduced to the inequivalence problem of FTAs. By the simulations given in [ChaKoSto81] the result follows.

Without loss of generality we assume that

- M has only one accepting state f and no transitions in state f;
- M has no negating states;
- in M, universal states have always exactly two successors.

Without loss of generality the work tape of M on a computation for w always contains N symbols. Therefore, configurations of M can be denoted by $w_1 \uparrow w_2 q$ where $w_1 w_2 \in \Gamma^N$ is the current inscription of the work tape, $\uparrow \notin \Gamma$ stands left to the position of the read/write head of M, and q represents the current state.

Define Σ by
$\Sigma_0 = \{f\}$,
$\Sigma_1 = \Gamma \cup \{\uparrow\} \cup \{q \mid q \text{ existential state of } M\}$,
$\Sigma_2 = \{q \mid q \text{ universal state of } M\}$, and
$\Sigma_m = \varnothing$ for all $m > 2$.

Note that every accepting computation tree of M for w can be represented by some $t \in T_\Sigma$.

We construct an FTA A such that $L(A) = \{t \in T_\Sigma \mid t \text{ is not an accepting computation tree for } w\}$. Given an input tree t, A nondeterministically performs one of the following tasks:

(1) Test whether t does not start with the initial configuration $\uparrow wq_0$ of a computation of M for w;

(2) Test whether t contains an "ill-formed configuration," i.e., whether t contains a sequence of unary symbols such that there are more or less than N symbols of Γ and more or less than one $\uparrow$ between two states;

(3) Test whether t contains an incorrect transition between two configurations, i.e., whether one of the tape symbols is not copied correctly, whether $\uparrow$ has moved more than one step, or whether there is no transition of M to cause the change of current input symbol, state, or head position;

(4) Test whether t contains a configuration of M with universal state such that the two following configurations in t do not correspond to two different transitions of M.

Each of these tasks can be implemented with $O(N)$ states. Since (a description of) A can be constructed from w in logspace, and from $\text{ASPACE}(n) = \bigcup_{c>0} \text{DTIME}(2^{cn})$ [ChaKoSto81], the result follows. □

THEOREM 2.2. *The inequivalence problem for* FTAs *accepting finite languages is* PSPACE*-complete with respect to* logspace *reductions.*

Similar to the proof of Theorem 2.1, the proof of Theorem 2.2 will be an appropriate generalization of a proof for the NP-completeness of deciding inequivalence of finite word automata accepting finite languages. In the case of finite word automata the bound on the length of the witness for inequivalence allows one to construct an NP-algorithm; in the case of tree automata the bound on the depth of a witness for inequivalence allows one to construct a polynomially space bounded algorithm. Accordingly for the hardness part, instead of satisfiability of conjunctive normal forms that can be encoded into the inequivalence problem for finite word automata accepting finite languages, we can encode arbitrarily quantified conjunctive normal forms into the inequivalence problem for finite tree automata accepting finite languages.

Before giving a detailed proof of Theorem 2.2, we state an immediate corollary.

COROLLARY 2.3. *The inequivalence problem for finitely ambiguous* FTAs *is* PSPACE*-hard.*

Proof. FTAs accepting finite languages form a subclass of the class of finitely ambiguous FTAs. □

Proof of Theorem 2.2. We need the notion of branches of trees. For a ranked alphabet Σ define Σ_B as the (ordinary) alphabet

$$\Sigma_B = \{(a, j) \mid a \in \Sigma_m, m > 0, j \in \{0, \cdots, m-1\}\}.$$

Assume $t \in T_\Sigma$, $r = j_1 \cdots j_k \in S(t)$ is a leaf of t, and $a_1, \cdots, a_k$ is the sequence of labels of the nodes on the path from the root of t to r (omitting the label of the leaf itself), i.e., $a_\kappa = \lambda_t(j_1 \cdots j_{\kappa-1})$ for $\kappa = 1, \cdots, k$. Then $(a_1, j_1) \cdots (a_k, j_k)$ is called the branch of t corresponding to r.

Now assume $A_i = (Q_i, \Sigma, Q_{i,0}, \delta_i)$, $i = 1, 2$, are two FTAs accepting finite languages, $\# Q_i \leqq n$ and $rk(\Sigma) = L$. The computations of $P(A_1)$ and $P(A_2)$ for a witness for inequivalence can be simulated by a nondeterministic Turing machine M that uses its worktape as a pushdown store (with entries of size $O(n)$ to hold pairs of sets of states). M accepts if and only if it has simulated correctly the computations of $P(A_1)$ and $P(A_2)$ on some t in the symmetric difference of $L(A_1)$ and $L(A_2)$. Since $L(A_1)$ and $L(A_2)$ are finite, depth $(t) < n$. Therefore, during an accepting computation of M, the pushdown store never contains more than $L \cdot n$ pairs of sets of states. It follows that M has only polynomial space complexity.

PSPACE-hardness: By [StoMe73] the following set **B** is known to be PSPACE-complete with respect to logspace reductions:

$\mathbf{B} = \{\langle x_1, y_1, \cdots, x_k, y_k, B\rangle \mid k \geqq 0, B$ conjunctive normal form containing variables from $x_1, y_1, \cdots, x_k, y_k$ such that $\exists x_1\, \forall y_1 \cdots \exists x_k\, \forall y_k\, B(x_1, y_1, \cdots, x_k, y_k)\}$.

Therefore, fix some $k \geqq 0$, pairwise different variables $x_1, \cdots, x_k, y_1, \cdots, y_k$ and a conjunctive normal form $B(x_1, y_1, \cdots, x_k, y_k)$. We want to construct FTAs A_1, A_2 accepting finite languages such that

$$L(A_1) \neq L(A_2) \quad \text{iff} \quad \langle x_1, y_1, \cdots, x_k, y_k, B\rangle \in \mathbf{B}.$$

Let Σ denote the ranked alphabet with $\Sigma_0 = \{\#\}$, $\Sigma_2 = \{0, 1\}$ and $\Sigma_j = \varnothing$ otherwise. Both the integers 0 and 1 and the corresponding two elements in Σ_2 will be used to represent the truth values "false" and "true," respectively.

Define a set $T^{(k)} \subseteq T_\Sigma$ inductively by

$$T^{(0)} = \{\#\}, \quad \text{and} \quad T^{(k)} = \{a(t_1, t_2) \mid a \in \Sigma_2, t_1, t_2 \in T^{(k-1)}\} \quad \text{for } k > 0.$$

Obviously, $depth(t) = k$ for every $t \in T^{(k)}$. Define an FTA A_2 of size $O(k)$ such that $T^{(k)} = L(A_2)$. Clearly, A_2 can be constructed by a logspace Turing machine.

We construct an FTA A_1 with $L(A_1) \subseteq T^{(k)}$ and $L(A_1) \neq T^{(k)}$ iff $\exists x_1\, \forall y_1 \cdots \exists x_k\, \forall y_k\, B(x_1, y_1, \cdots, x_k, y_k)$.

Given an input tree t, A_1 behaves as follows:

(1) A_1 checks whether t is in $T^{(k)}$ and rejects any other tree. So, without loss of generality assume $t \in T^{(k)}$.

(2) A_1 guesses a clause c of B and a branch $(\xi_1, \eta_1) \cdots (\xi_k, \eta_k)$ and accepts whenever

$$c(\xi_1, \eta_1, \cdots, \xi_k, \eta_k) = 0.$$

A_1 can be constructed in logspace as well.

For the correctness of the construction we observe that $\exists x_1\, \forall y_1 \cdots \exists x_k\, \forall y_k\, B(x_1, y_1, \cdots, x_k, y_k)$ if and only if there is a tree $t \in T^{(k)}$ such that t satisfies (*):

(*) $\quad B(\xi_1, \eta_1, \cdots, \xi_k, \eta_k) = 1$ for all branches $(\xi_1, \eta_1) \cdots (\xi_k, \eta_k)$ of t.

However, by the construction of A_1, A_1 accepts exactly those trees in $T^{(k)}$ that do not satisfy (*). Thus,

$$L(A_1) \neq L(A_2) \quad \text{iff } \exists x_1\, \forall y_1 \cdots \exists x_k\, \forall y_k\, B(x_1, y_1, \cdots, x_k, y_k). \qquad \square$$

3. Semiring automata. We extend our notion of a finite tree automaton by allowing the transitions and initial states to have weights in some semiring R. The advantage of this extension is twofold.

On the one hand the resulting automata have nice algebraic properties that make it possible to "eliminate" finite ambiguity. These properties are studied in this section. On the other hand, the extension enables us to use methods from linear algebra to decide ambiguity-equivalence. This will be investigated in the next section.

A commutative semiring with 0 and 1 is a structure $(R, +, \cdot)$, where 0 and 1 are two different elements of R; $+$ and $\cdot$ are commutative and associative operations on R; and the following equations hold for arbitrary $a, b, c \in R$:

$$a \cdot (b+c) = (a \cdot b) + (a \cdot c), \qquad 1 \cdot a = a, \qquad 0 + a = a, \qquad 0 \cdot a = 0.$$

If $+$ and $\cdot$ are understood, we write R instead of $(R, +, \cdot)$. Moreover, since all semirings we will use are commutative and have 0 and 1, we will call them semirings for short. Assume R is such a semiring. A finite tree automaton with weights in R (short: R-FTA) is a quadruple $A = (Q, \Sigma, I, \delta)$, where

Q is a finite set of states;

Σ is a ranked alphabet;

$I = (I_q)_{q \in Q} \in R^Q$ is the Q-tuple of initial ambiguities; and

δ is a map $\delta : \bigcup_{m \geq 0} Q \times \Sigma_m \times Q^m \to R$ denoting the transition multiplicites.

Let $V = R^Q$ denote the set of Q-tuples of semiring elements. Taking equations (1) and (2) of Proposition 1.1 as a definition, we define

(1) a map $n_A : T_\Sigma \to V$ by $n_A(t) = (n_A(t))_{q \in Q}$, where

$$n_A(t)_q = \sum_{q_0 \cdots q_{m-1} \in Q^m} \delta(q, a, q_0, \cdots, q_{m-1}) \cdot n_A(t_0)_{q_0} \cdot \cdots \cdot n_A(t_{m-1})_{q_{m-1}}$$

for $t = a(t_0, \cdots, t_{m-1})$, and

(2) a map $da_A : T_\Sigma \to R$ by $da_A(t) = \sum_{q \in Q} I_1 \cdot n_A(t)_q$.

$n_A(t)$ is called ambiguity vector of A for t in V; $da_A(t)$ is called the ambiguity of A for t in R. Note that by (1), every $a \in \Sigma_m$ defines a multilinear map $a : V^m \to V$ which in particular yields, when applied to the ambiguity vectors of A for the subtrees t_j, $j = 0, \cdots, m-1$, the ambiguity vector of A for $a(t_0, \cdots, t_{m-1})$.

Finally, the language $L(A)$ accepted by A is defined by $L(A) = \{t \in T_\Sigma \mid da_A(t) \neq 0\}$. Similar to the case of FTAs, the size of the R-FTA A, $|A|$, is defined by

$$|A| = \sum_{\delta(q,a,q_0 \cdots q_{m-1}) \neq 0} (m+2).$$

An R-FTA A is called unambiguous if and only if $da_A(t) \in \{0, 1\}$ for all $t \in T_\Sigma$.

Two R-FTAs A_1, A_2 are called equivalent if and only if $L(A_1) = L(A_2)$. They are called ambiguity-equivalent (denoted: $A_1 \equiv A_2$) if and only if $da_{A_1}(t) = da_{A_2}(t)$ for every tree t.

An FTA $A = (Q, \Sigma, Q_I, \delta)$ can be viewed as the definition of an R-FTA $A_R = (Q, \Sigma, 1_{Q_I}, \delta_R)$, where

$$1_{Q_I, q} = \begin{cases} 1 & \text{if } q \in Q_I \\ 0 & \text{else} \end{cases}$$

and

$$\delta_R(q, a, q_0 \cdots q_{m-1}) = \begin{cases} 1 & \text{if } (q, a, q_0 \cdots q_{m-1}) \in \delta \\ 0 & \text{else.} \end{cases}$$

We make the following observations.

PROPOSITION 3.1. *For all $t \in T_\Sigma$:*

(1) *If $\mathbb{N}_0$ is a subsemiring of R, then*

(1.1) $n_{A_R}(t)_q = n_A(t)_q$ *for all* $q \in Q$;

(1.2) $daA_R(t) = da_A(t)$.

(2) *If $p > 1$ is a natural number, then*

(2.1) $(n_{A_{\mathbb{Z}_p}}(t)_q = n_A(t)_q \bmod p)$ *for all* $q \in Q$;

(2.2) $da_{A_{\mathbb{Z}_p}}(t) = da_A(t) \bmod p$.

(3) *If $\mathbb{B}$ is the Boolean semiring defined by $1+1=1$, then*

(3.1) $(n_{A_{\mathbb{B}}}(t)_q = 1 \quad \text{iff} \quad n_A(t)_q > 0)$ *for all* $q \in Q$;

(3.2) $da_{A_{\mathbb{B}}}(t) = 1 \quad \text{iff} \quad da_A(t) > 0$.

Proof. In all three cases there are semiring homomorphisms $\rho_R : \mathbb{N}_0 \to R$ with $\rho_R(0) = 0$ and $\rho_R(1) = 1$. Thus, the subcases (_.2) follow directly from the subcases (_.1), and the subcases (_.1) follow by induction on the depth of t. □

For convenience we no longer distinguish between an FTA A and its corresponding $\mathbb{N}_0$-FTA $A_{\mathbb{N}_0}$.

Let R denote a semiring. The next three lemmas provide constructions for linear combination of R-FTAs, products of R-FTAs, and constant R-FTAs.

Assume $A_i = (Q_i, \Sigma, I^{(i)}, \delta_i)$, $i = 1, 2$, are two R-FTAs and $\mu_1, \mu_2 \in R$. By $\mu_1 A_1 + \mu_2 A_2$ we denote the R-FTA $(\bar{Q}, \Sigma, \bar{I}, \bar{\delta})$, where $\bar{Q}$ is the disjoint union of Q_1 and Q_2;

$$\bar{I}_q = \begin{cases} \mu_1 I_q^{(1)} & \text{if } q \in Q_1 \\ \mu_2 I_q^{(2)} & \text{if } q \in Q_2 \end{cases}$$

$$\bar{\delta}(q, a, q_0 \cdots q_{m-1}) = \begin{cases} \delta_1(q, a, q_0 \cdots q_{m-1}) & \text{if } q, q_0, \cdots, q_{m-1} \in Q_1 \\ \delta_2(q, a, q_0 \cdots q_{m-1}) & \text{if } q, q_0, \cdots, q_{m-1} \in Q_2 \\ 0 & \text{else} \end{cases}$$

By $A_1 \times A_2$ we denote the R-FTA $(\bar{\bar{Q}}, \Sigma, \bar{\bar{I}}, \bar{\bar{\delta}})$, where

$\bar{\bar{Q}} = Q_1 \times Q_2$;

$\bar{\bar{I}}_{(p,q)} = I_p^{(1)} \cdot I_q^{(2)}$ for $p \in Q_1$, $q \in Q_2$; and

$\bar{\bar{\delta}}((p, q), a, (p_0, q_0) \cdots (p_{m-1}, q_{m-1})) = \delta_1(p, a, p_0 \cdots p_{m-1}) \cdot \delta_2(q, a, q_0 \cdots q_{m-1})$.

Finally, for every $j \in R$ we define the constant R-FTA

$$\mathbf{j} = (\{q\}, \Sigma, j, \delta^1), \quad \text{where } \delta^1(q, a, q^m) = 1 \text{ for all } a \in \Sigma_m,\ m \geqq 0.$$

PROPOSITION 3.2. *For all $t \in T_\Sigma$:*

(1) $$n_{\mu_1 A_1 + \mu_2 A_2}(t)_q = \begin{cases} n_{A_1}(t)_q & \text{if } q \in Q_1 \\ n_{A_2}(t)_q & \text{if } q \in Q_2 \end{cases}$$

(2) $$da_{\mu_1 A_1 + \mu_2 A_2}(t) = \mu_1 da_{A_1}(t) + \mu_2 da_{A_2}(t).$$ □

PROPOSITION 3.3. *For all $t \in T_\Sigma$:*

(1) $n_{A_1 \times A_2}(t)_{(p,q)} = n_{A_1}(t)_p \cdot n_{A_2}(t)_q$ *for all* $p \in Q_1$ *and* $q \in Q_2$;

(2) $da_{A_1 \times A_2}(t) = da_{A_1}(t) \cdot da_{A_2}(t)$. □

PROPOSITION 3.4. *For all $t \in T_\Sigma$:*

(1) $n_{\mathbf{j}}(t) = n_{\mathbf{j}}(t)_q = 1$

(2) $da_{\mathbf{j}}(t) = j$. □

The proofs of the Propositions 3.2–3.4 are omitted. Their assertions (1) can be verified by induction on the structure of terms, whereas assertions (2) immediately follow from the definitions and the assertions (1).

In [Kui88] Kuich describes a transformation transforming an m-ambiguous finite word automaton into an unambiguous $\mathbb{Q}$-automaton A' such that $L(A) = L(A')$. The

only properties that are needed are the presence of constructions for the linear combination and product of automata (as we have proved to exist for R-FTAs as well in 3.2 and 3.3) and the existence of automata accepting every word with a fixed ambiguity (similar to the R-FTA's **j**). Thus we get Proposition 3.5.

PROPOSITION 3.5. *Assume A is an m-ambiguous* FTA. *Then*

$$un(A) = \sum_{j=1}^{m} \frac{(-1)^{j+1}}{j!}[A]_j$$

is an unambiguous $\mathbb{Q}$-FTA *where* $[A]_j = A \times (A-1) \times \cdots \times (A-(\mathbf{j}-1))$. *Furthermore,* $L(un(A)) = L(A)$.

Proof. If $k = da_A(t) > 0$, then $da_{[A]_j}(t) = k \cdot (k-1) \cdot \cdots \cdot (k-j+1)$, and hence

$$da_{un(A)}(t) = \sum_{j=1}^{m} (-1)^{j+1}\binom{k}{j} = \sum_{j=1}^{k} (-1)^{j+1}\binom{k}{j} = 1.$$

If $da_A(t) = 0$, then $da_{[A]_j}(t) = 0$ for all $j > 0$, and hence also $da_{un(A)}(t) = 0$. □

Note that $un(A)$ can be constructed on a Random Access Machine (RAM) (without multiplications) in time $O(|A|^m)$.

We relativize the construction of un (A) modulo a suitable prime number p. The reason for this is to keep the numbers occurring in our algorithms small. Assume p is a prime number greater than m. By Bertrand's postulate (see [HaWri60, Thm. 418]) such a prime p exists in the range between m and $2m$. Since $p > m$, the multiplicative inverse $(j! \bmod p)^{-1}$ is defined in $\mathbb{Z}_p$ for all $j \in \{1, \cdots, m\}$. Therefore, we can define a $\mathbb{Z}_p$-FTA $un(A)_p$ by

$$un(A)_p = \sum_{j=1}^{m} (-1)^{j+1}(j! \bmod p)^{-1}[A_{\mathbb{Z}_p}]_j.$$

As a consequence of Proposition 3.1(2) and Proposition 3.5 we get the following theorem.

THEOREM 3.6. *For every m-ambiguous* FTA, A, *and every prime number $p > m$, $un(A)_p$ is an unambiguous $\mathbb{Z}_p$-*FTA *with $L(A) = L(un(A)_p)$.*

Since for unambiguous R-FTAs equivalence coincides with ambiguity-equivalence, Theorem 3.6 can be used to reduce the equivalence problem for m-ambiguous FTAs to the ambiguity-equivalence problem for unambiguous $\mathbb{Z}_p$-FTAs.

4. Deciding ambiguity-equivalence. In this section we present an algorithm deciding ambiguity-equivalence for R-FTAs, provided R is a field. This algorithm relies on a generalization of Eilenberg's equality theorem [Ei74, Thm. 8.1] to finite tree automata.

MAIN LEMMA 4.1. *If R is a field, and A is an R-*FTA *with n states, then the following holds.*

(1) $A \equiv \mathbf{0}$ iff $da_A(t) = 0$ for all $t \in T_\Sigma$ with $depth(t) < n$.

(2) Whether or not $A \equiv \mathbf{0}$ can be decided in polynomial time on a RAM which is allowed to hold elements of R in its registers and to perform the R-operations $+$, $-$, $\cdot$, : and R-tests for 0 in constant time.

As an immediate consequence we get Theorem 4.2.

THEOREM 4.2. *If R is a field, and A_1 and A_2 are R-*FTA*s with n_1 and n_2 states, respectively, then the following holds.*

(1) $A_1 \equiv A_2$ iff $da_{A_1}(t) = da_{A_2}(t)$ *for all* $t \in T_\Sigma$ *with* $depth(t) < n_1 + n_2$.

(2) *Whether $A_1 \equiv A_2$ can be decided in polynomial time on a* RAM *which can hold elements of R in its registers and performs the R-operations $+$, $-$, $\cdot$, : and R-tests for* 0 *in constant time.*

Proof. $A_1 \equiv A_2$ iff $A_1 - A_2 \equiv 0$. Thus, Theorem 4.2 follows from the Main Lemma 4.1. □

Proof of 4.1. Assume $A = (Q, \Sigma, \delta, I)$ and $rk(A) = L$. Let V denote the n-dimensional R-vector space $V = R^Q$. For $k \geqq 0$ define $V_k = \langle n_A(t) \mid depth(t) \leqq k \rangle$, i.e., V_k is the subspace of V generated by the ambiguity vectors of trees t with $depth(t) \leqq k$. Clearly, $V_0 \subseteq V_1 \subseteq \cdots \subseteq V_k \subseteq V_{k+1} \cdots \subseteq V$.

Recall that every $a \in \Sigma$ defines a multilinear map $a: V^m \to V$; and we have

$$V_{k+1} = \left\langle \bigcup_{a \in \Sigma} a(V_k, \cdots, V_k) \right\rangle.$$

We conclude

(i) If $V_k = V_{k+1}$ for some $k \geqq 0$, then $V_k = V_{k+1}$ for all $1 > 0$; and therefore, since $\dim(V) = n$

(ii) $V_{n-1} = V_n = \bigcup_{k \geqq 0} V_k$.

(iii) If B_k is a basis of V_k, then $B'_{k+1} = B_k \cup \bigcup_{m>0} \{a(v_0, \cdots, v_{m-1}) \mid a \in \Sigma_m, v_j \in B_k\}$ is a generating system for V_{k+1}.

(iv) The following three statements are equivalent:

(a) $\sum_{q \in Q} I_q n_A(t)_q = 0$ for all $t \in T_\Sigma$;

(b) $\sum_{q \in Q} I_q v_q = 0$ for all $v = (v_q)_{q \in Q} \in V_{n-1}$;

(c) $\sum_{q \in Q} I_q v_q = 0$ for all $v = (v_q)_{q \in Q} \in B_{n-1}$ of some basis B_{n-1} of V_{n-1}.

By (iii) there is a basis B_{n-1} of V_{n-1} consisting only of vectors $n_A(t)$ for some $t \in T_\Sigma$ of depth less than n. This proves (1).

Note that this proof results from Eilenberg's proof by using multilinear maps instead of linear maps.

Ad(2): For one $k > 0$, computing B'_k from B_k needs $O(\#\Sigma \cdot n^L \cdot |A|)$ operations ($\#\Sigma$ denotes the number of elements of Σ). B'_k contains $O(n^L \cdot \#\Sigma)$ elements. By the Gaussian elimination method we can compute from B'_k a basis for V_k in $O(n^{L+2} \cdot \#\Sigma)$ steps. Without loss of generality we may assume $n \leqq |A|$. Thus, a basis for V_{n-1} can be computed in time $O(n^{L+2}|A| \cdot \#\Sigma)$. Since testing whether $\sum_{q \in Q} I_q v_q = 0$ for all $v = (v_q)_{q \in Q} \in B_{n-1}$ can be done in time $O(n^2)$, this is already the final complexity. By Proposition 1.3 we may assume $L \leqq 2$. Hence, assertion (2) follows. □

Note that the RAM performing our test for $A \equiv \mathbf{0}$ makes intensive use of unrestricted multiplication. Especially, if A is an FTA the entries of the basis vectors to be considered may grow very rapidly. The only upper bound for these entries given by the algorithm is $n^{L+L^2+\cdots+L^n}$. Thus, for $L > 1$ the occurring integers may have length $O(L^n \cdot \log n)$. However, we can restrict the lengths of occurring numbers by employing the fields $\mathbb{Z}_p$ (p a prime number) instead of $\mathbb{Q}$ as domains for the weights of the FTAs in question. Thus, for testing equivalence of m-ambiguous FTAs we can construct a deterministic polynomial algorithm; whereas for testing ambiguity-inequivalence of arbitrary FTAs we are able to give at least a randomized polynomial algorithm.

THEOREM 4.3. *The equivalence problem for m-ambiguous* FTA*s is* P*-complete with respect to* logspace *reductions for every fixed constant* $m > 0$.

Proof. Assume A_i, $i = 1, 2$, are two m-ambiguous FTAs. By Theorem 3.6 it suffices to decide whether $un(A_1)_p \equiv un(A_2)_p$ for a prime number $p > m$. The $\mathbb{Z}_p$-FTAs $un(A_i)_p$,

$i = 1, 2$, can be constructed in polynomial time, and the algorithm of Theorem 4.2(2) needs only the $\mathbb{Z}_p$-operations $+, -, \cdot, :$. Therefore, deciding equivalence of m-ambiguous FTAs is in P. Since the emptiness problem for context-free grammars can be reduced in logspace to the equivalence problem even of unambiguous FTAs, the result follows. □

For the following proposition, again let A_1, A_2 denote two FTAs with rank L and $\leqq n$ states. To avoid trivial subcases, we assume $n > 1$. For $k > 0$ let p_k denote the kth prime number.

PROPOSITION 4.4. *Define* $K = \log(n) \cdot (L+1)^{2n}$ (*where* $\log(\_)$ *denotes the logarithm with base* 2).

(1) $A_1 \equiv A_2$ iff $(A_{1,\mathbb{Z}_{p_k}} \equiv A_{2,\mathbb{Z}_{p_k}}$ *for all* $k \leqq K)$.

(2) *If* $K_0 \geqq 2K$ *and* $p \in \{p_1, \cdots, p_{K_0}\}$ *is randomly chosen* (*with respect to the equal distribution*) *then*

(2.1) *if* $A_1 \equiv A_2$ *then* $\text{prob}\{A_{1,\mathbb{Z}_p} \equiv A_{2,\mathbb{Z}_p}\} = 1$; *and*

(2.2) *if* $A_1 \not\equiv A_2$ *then* $\text{prob}\{A_{1,\mathbb{Z}_p} \not\equiv A_{2,\mathbb{Z}_p}\} \geqq \frac{1}{2}$.

Proof:

Ad(1): If $A_1 \equiv A_2$, then by Proposition 3.1(2) $A_{1,\mathbb{Z}_p} \equiv A_{2,\mathbb{Z}_p}$. Now assume A_1 and A_2 are not ambiguity-equivalent. By Theorem 4.2 there is some $t \in T_\Sigma$ with $depth(t) < 2n$ such that $da_{A_1}(t) \neq da_{A_2}(t)$. The bound on the depth of t implies that $da_{A_i}(t) \leqq n^{1+L+\cdots+L^{2n}} < n^{(L+1)^{2n}}$. Since $\prod_{k \leqq K} p_k \geqq 2^K = n^{(L+1)^{2n}}$, it follows from the Chinese Remainder Theorem that a prime number $p \in \{p_k \mid k \leqq K\}$ exists such that $da_{A_1}(t) \bmod p \neq da_{A_2}(t) \bmod p$, and therefore $A_{1,\mathbb{Z}_p} \not\equiv A_{2,\mathbb{Z}_p}$.

Ad(2): Assertion (2.1) is again the immediate consequence of Proposition 3.1(2). By Theorem 4.2 there is some $t \in T_\Sigma$ such that $z := da_{A_1}(t) - da_{A_2}(t) \neq 0$ and $|z| \leqq n^{(L+1)^{2n}}$. Since z contains at most K primes as a factor, assertion (2.2) follows. □

THEOREM 4.5. *The ambiguity-inequivalence problem for* FTA*s is in* RP, *i.e., the class of problems with randomized polynomial algorithms.*

Proof. Assume $A_i = (Q_i, \Sigma, Q_{i,I}, \delta_i)$, $i = 1, 2$, are two FTAs with $\# Q_i \leqq n$. By Proposition 1.3 we may assume without loss of generality that $L = \max\{rk(a) \mid a \in \Sigma\} \geqq 2$.

Define $K = \log(n) \cdot (L+1)^{2n}$ as in Proposition 4.4. Since $p_k = O(k \cdot \log(k))$ (see [Ap76] or [RoSchoe62]), one can choose constants $c, c' > 0$ such that $p_{\lceil 2K \rceil} < 2^{cn}$ and the cardinality of the set $\{p \mid p \text{ prime}, p < 2^{cn}\}$ is at least $c'(2^{cn}/n)$. We construct a probabilistic polynomial algorithm **A** that on input A_1, A_2 behaves as follows:

(i) If $A_1 \equiv A_2$, then prob{**A** outputs: "ambiguity-equivalent"} = 1;

(ii) If $A_1 \not\equiv A_2$, then prob{**A** outputs: "not ambiguity-equivalent"} $\geqq c'/2n$.

If we repeat this algorithm N times, where $N \geqq 2n/c'$, we get an algorithm **A'** that satisfies (i) but outputs: "not ambiguity-equivalent" with probability $\geqq \frac{1}{2}$ if $A_1 \not\equiv A_2$. The algorithm **A** behaves as follows:

(1) **A** guesses a nonnegative integer $p \in \{0, \cdots, 2^{cn} - 1\}$.

(2) **A** constructs the $\mathbb{Z}_p$-FTA $\bar{A} = A_{1,\mathbb{Z}_p} - A_{2,\mathbb{Z}_p}$.

(3) **A** tries to compute the linearly independent set of vectors $B_{2n-1} \subseteq \{n_{\bar{A}}(t) \mid t \in T_\Sigma\}$ according to the algorithm given in the proof of 4.1(2).

If the multiplicative inverse of some $q \in \mathbb{Z}_p$ has to be computed, then A tests whether $q^{p-1} = 1$.

If $q^{p-1} \neq 1$, then **A** outputs: "ambiguity-equivalent".

If $q^{p-1}=1$, then **A** computes q^{p-2} which in this case is the multiplicative inverse of q.

(4) **A** computes $z(v) := \sum_{q\in Q_{1,l}} v_q - \sum_{q\in Q_{2,l}} v_q$ for all $v \in B_{2n-1}$. If $z(v)=0$ for all $v \in B_{2n-1}$, then **A** outputs "ambiguity-equivalent". If not, then **A** outputs: "not ambiguity-equivalent."

Clearly, this algorithm runs in polynomial time. We show that **A** has the properties (i) and (ii).

Assume $A_1 \equiv A_2$. We distinguish two cases:

Case I. **A** succeeds to compute B_{2n-1}. $B_{2n-1} \subseteq \{n_A(t) \mid t \in T_\Sigma\}$. Therefore, by Propositions 3.1-3.2 we have for all $v \in B_{2n-1}$ $\exists t \in T_\Sigma$:

$$\begin{aligned} z(v) &= da_{\bar{A}}(t) \\ &= (da_{A_1}(t) - da_{A_2}(t)) \bmod p \\ &= 0. \end{aligned}$$

Hence, **A** outputs "ambiguity-equivalent."

Case II. **A** does not succeed to compute B_{2n-1}. Then **A** always outputs "ambiguity-equivalent." Therefore, **A** has property (i).

Now assume $A_1 \not\equiv A_2$. Assume $p \in \{0, \cdots, 2^{cn}-1\}$ is the integer randomly chosen in step (1). Then p is a prime number with probability $\geqq c'/n$. If p is a prime number, then $\mathbb{Z}_p$ is a field and hence the algorithm in the proof of 4.1(2) works correctly. Especially, for every $q \in \mathbb{Z}_p$, $q^{p-1}=1$ and q^{p-2} is the multiplicative inverse of q; thus **A** outputs "not ambiguity-equivalent," if and only if $A_{1,\mathbb{Z}_p} \not\equiv A_{2,\mathbb{Z}_p}$. By Proposition 4.4(2), $A_{1,\mathbb{Z}_p} \not\equiv A_{2,\mathbb{Z}_p}$ with probability $\geqq \frac{1}{2}$. Therefore, **A** outputs "not ambiguity-equivalent" with probability $\geqq c'/2n$. Hence, **A** has also property (ii). This finishes the proof. □

5. Finite degree of ambiguity. We have seen that bounding the degree of ambiguity by some fixed constant m yields a class of FTAs with a polynomial equivalence problem. Therefore, for the sake of completeness we show in the following theorem:

THEOREM 5.1. *For every* FTA A and $m>0$ *the following holds.*

(1) *There is an* FTA $A^{(m)} = (Q^{(m)}, \Sigma, Q_I^{(m)}, \delta^{(m)})$ *such that* $L(A^{(m)}) = \{t \in T_\Sigma \mid da_A(t) \geqq m\}$. *Especially,* $L(A^{(m)}) = \emptyset$ iff $da(A) < m$.

(2) *If m is a fixed constant, then it can be decided in polynomial time whether or not* $da(A) < m$.

Proof. Assume $A = (Q, \Sigma, Q_I, \delta)$, where $\# Q = n$ and $rk(A) = L$. Let $[0, m]$ denote the set $\{0, 1, \cdots, m\}$. Consider the semiring $([0, m], (+), *)$ with "absorbing upper bound," i.e., $(+)$ and $*$ are defined by

$$m_1 (+)\, m_2 = \begin{cases} m & \text{if } m_1 + m_2 \geqq m \\ m_1 + m_2 & \text{else} \end{cases}$$

and

$$m_1 * m_2 = \begin{cases} m & \text{if } m_1 \cdot m_2 \geqq m \\ m_1 \cdot m_2 & \text{else} \end{cases}$$

Define $Q^{(m)} = \{v \in [0, m]^Q \mid \#\{q \mid v_q \neq 0\} \leqq m\}$.

(As usual, for a Q-tuple v we adopt the convention that v_q denotes the qth element in v.)

Define $Q_I^{(m)} = \{v \in Q^{(m)} \mid (+)_{q\in Q_I} v_q = m\}$.

If $m \geqq n$, define $\delta^{(m)}$ as follows. For every $a \in \Sigma_k$ and $v_0, \cdots, v_{k-1} \in [0, m]^Q$, $(v, a, v_0, \cdots, v_{k-1})$ is in $\delta^{(m)}$ iff $v_q = (+)_{(q,a,q_0 \cdots q_{k-1}) \in \delta} v_{0,q_0} * \cdots * v_{k-1,q_{k-1}}$ for all $q \in Q$.

If $m < n$, define $\delta^{(m)}$ as follows. For every $a \in \Sigma_k$, $\delta' \subseteq \delta$, with $\#\delta' \leqq m$ and $v_1, \cdots, v_k \in Q^m$, $(v, a, v_0 \cdots v_{k-1})$ is in $\delta^{(m)}$ if and only if $v_q = (+)_{(q,a,q_0 \cdots q_{k-1}) \in \delta'} v_{0,q_0} * \cdots * v_{k-1,q_{k-1}}$ for all $q \in Q$. By the restriction to the cardinality of δ', $\#\{q \mid v_q \neq 0\} \leqq m$.

We omit the proof that the above constructed automaton has the property stated in assertion (1).

If $m < n$ is a fixed constant, then $A^{(m)}$ can be constructed in time $O(|A|^m \cdot m^{mL})$. Recall that an FTA accepts the empty set if and only if the corresponding reduced FTA has an empty state set. By Proposition 1.2 the reduced FTA for $A^{(m)}$ can be constructed in time $O(|A^{(m)}|) \leqq O(|A|^m \cdot m^{mL})$. Therefore, it can be decided in time $O(|A|^m \cdot m^{mL})$, whether $L(A^{(m)})$ is empty or not. Since we may assume $L \leqq 2$, it can be decided in polynomial time whether or not $da(A) < m$. □

Acknowledgment. I thank Andreas Weber for many fruitful discussions and for carefully reading an earlier version of this paper.

REFERENCES

[Aho74] A. V. AHO, J. E. HOPCROFT, AND J. D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[Ap76] T. APOSTOL, *Introduction to Analytic Number Theory*, Springer-Verlag, New York, Berlin, 1976.

[BeReu82] J. BERSTEL AND C. REUTENAUER, *Recognizable formal power series on trees*, TCS, 18 (1982), pp. 115-148.

[ChaKoSto81] A. K. CHANDRA, D. C. KOZEN, AND L. J. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[Do70] J. DONER, *Tree acceptors and some of their applications*, J. Comput. System Sci., 4 (1970), pp. 406-451.

[Ei74] S. EILENBERG, *Automata, Languages, and Machines*, Vol. A, Academic Press, New York, 1974.

[GeStei84] F. GECSEG AND M. STEINBY, *Tree automata*, Akad. Kiado, Budapest, 1984.

[HaWri60] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, 4th ed., Oxford University Press, London, 1960.

[Kui88] W. KUICH, *Finite automata and ambiguity*, Technische Universitaet Graz und österreichische Computer Gesellschaft, Report 253, June 1988.

[MeSto72] A. R. MEYER AND L. J. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential space*, 13th SWAT, 1972, pp. 125-129.

[Paul78] W. J. PAUL, *Komplexittstheorie*, B. G. Teubner, Stuttgart, 1978.

[Ra69] M. O. RABIN, *Decidability of second-order theories and automata on infinite trees*, Trans. Amer. Math. Soc., 141 (1969), pp. 1-35.

[RoSchoe62] B. ROSSER AND L. SCHOENFELD, *Approximate formulas for some functions of prime numbers*, Illinois J. of Math., 6 (1962), pp. 64-94.

[Se89] H. SEIDL, *On the finite ambiguity of finite tree automata*, Acta Informatica, 26 (1989), pp. 527-542.

[SteHu81] R. STEARNS AND H. HUNT III, *On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata*, 22th FOCS, 1981, pp. 74-81.

[SteHu85] ———, *On the equivalence and containment problems for unambiguous regular expressions, regular grammars and finite automata*, SIAM J. Comput., 14 (1985), pp. 598-611.

[StoMe73] L. J. STOCKMEYER AND A. R. MEYER, *Word problems requiring exponential time*, 5th ACM-STOC, pp. 1-9, 1973.

[ThaWri68] J. W. THATCHER AND J. B. WRIGHT, *Generalized finite automata theory with an application to a decision problem of second order logic*, Math. Systems Theory, 2 (1968), pp. 57-81.

[Tho84] W. THOMAS, *Logical aspects in the study of tree languages*, in 9th Coll. on Trees in Algebra and Programming, B. Courcelle, ed., Cambridge University Press, 1984, pp. 31-49.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 438-451, June 1990

003

A NONTRIVIAL LOWER BOUND FOR AN NP PROBLEM ON AUTOMATA*

ETIENNE GRANDJEAN†

Abstract. An NP problem L is linearly NP-complete if each NTIME (n)-problem is reducible to L in linear time on a deterministic Turing machine. This implies that $L \notin$ DTIME (cn) for each $c \geqq 1$. Let R.I.S.A. (Reduction of Incompletely Specified Automata) be the following NP-complete problem (quoted AL7 in the classical book [M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman, San Francisco, 1979]. INSTANCE: a positive integer K and an incompletely specified deterministic finite state automaton $A = (Q, \Sigma, \delta, q_0, F)$, where δ is a "partial" transition function from $Q \times \Sigma$ into Q, and Q, Σ, q_0, F are defined as usual. QUESTION: Can δ be extended to a total transition function from $Q \times \Sigma$ into Q in such a way that the resulting completely specified automaton has an equivalent "reduced automaton" with K or fewer states?

It is proved that problem R.I.S.A. is linearly NP-complete. The proof uses a notion of generalized spectrum of a first-order sentence, which has the form $\forall y \bigwedge_{i<p} \mathcal{F}_i(y) = \mathcal{G}_i(y)$ where each $\mathcal{F}_i$, $\mathcal{G}_i$ is a word of the form $f_k \cdots f_2 f_1$, $k \geqq 0$, and each f_j is a unary function symbol.

Key words. NP-complete problem, Turing machine, random access machine, linear time reduction, spectrum of first-order sentence, reduction of automata, incompletely specified automaton.

AMS(MOS) subject classification. 68Q

1. Introduction.

In 1982, Cook [Co2] noticed that none of the many classical NP-complete problems (listed in [GaJo]) had a known time lower bound at this moment. In the present paper as in a previous one [Gr3], we present a *linearly* NP-*complete* problem (a problem is linearly NP-complete if each problem of NTIME (n) is reducible to it in linear time [De]). From a result of [PPST]:

$$\bigcup_{c \geqq 1} \text{DTIME}(cn) \subsetneqq \text{NTIME}(n).$$

It follows that such a problem does not belong to DTIME (cn) for any $c \geqq 1$.

The linearly NP-complete problem defined and studied in [Gr3] is a satisfiability problem (denoted $\text{SAT}_<(\mathbb{N})$) for a simple class of formulas interpreted over integers. We feel that $\text{SAT}_<(\mathbb{N})$ is a natural problem but we may object that it has been exhibited and studied because of its linear NP-completeness.

The present paper studies an NP-complete problem known and studied for a long time [PaUn], [Ko], [Pf], [ReMe] and listed in [GaJo, problem AL7]. This is the Reduction of Incompletely Specified Automata (R.I.S.A.).

INSTANCE. An incompletely specified deterministic finite state automaton $A = (Q, \Sigma, \delta, q_0, F)$, where Q is the set of states, Σ is the input alphabet, δ is a "partial" transition function from $Q \times \Sigma$ into Q, $q_0 \in Q$ is the initial state, and $F \subseteq Q$ is the set of "accept" states, and a positive integer K.

QUESTION. Can the transition function δ be extended to a total function from $Q \times \Sigma$ into Q in such a way that the resulting completely specified automaton recognizes the same language as a finite automaton with K or fewer states?

THEOREM 1.1. *Problem* R.I.S.A. *is linearly* NP-*complete.*

As in [Gr3] the proof of Theorem 1.1 uses a (rather technical) simulation of a nondeterministic Turing machine (NTM) by a Nondeterministic Random Access

* Received by the editors July 21, 1987; accepted for publication (in revised form) August 23, 1989.
† Laboratoire d'Informatique, Université de Caen, 14032 Caen Cedex, France.

Machine (NRAM). More essentially, our proof uses the (*finite*) *generalized spectrum* (see [Fa]) of a first-order sentence Φ (with only one variable y) of the form:

$$\Phi \equiv \forall y \bigwedge_{i<p} \mathscr{F}_i(y) = \mathscr{G}_i(y)$$

where each $\mathscr{F}_i$, $\mathscr{G}_i$ is a word of the form $f_k \cdots f_2 f_1$, with $k \geqq 0$, and each f_j is a unary function symbol. The notion of generalized spectrum with *bounded resources* (bounded number of quantifiers and bounded arity) was studied in several papers [Ly], [Gr1], [Gr2], [Gr3], [Gr4]. We are convinced that the notion of *generalized spectrum* (used for simulation of NRAMs) is the right one to prove the linear NP-completeness of natural problems.

Despite the superficial feeling that finite first-order structures are hard to manipulate, we think that they have a great expressiveness: in [Gr4] we exhibit some first-order sentences with only one variable which define some arithmetical functions on finite structures; in the present paper we obtain a very simple sentence (cf. Φ above) by using some *predefined* arithmetical functions on the universe of each finite structure.

Our paper is divided into seven sections. Section 2 presents some preliminaries. Because of the length and technicality of the proof of Theorem 1.1, § 3 gives a simplified idea of its proof, which is expounded in §§ 4, 5, and 6. Section 4 recalls some definitions and a result of [Mo], [Gr3] on an efficient simulation of NTMs by NRAMs. Section 5 presents the simulation of an NRAM by a first-order generalized spectrum with only one variable (the proof is omitted but can be found in [Gr3]); moreover, we prove that the first-order sentence can be reduced to the simple form Φ above. Section 6 gives an efficient reduction of the generalized spectrum of Φ to problem R.I.S.A.: this achieves the proof of Theorem 1.1. Section 7 presents some concluding remarks.

In the present paper all the stated results are proved, excepting Lemmas 4.1 and 5.1, which are proved in our previous paper [Gr3] (however, § 3 below gives an intuitive idea of their proofs). Therefore we suggest that the reader first reads [Gr3], which is self-contained.

2. Preliminaries. We use the usual notation and definitions in computational complexity (see [HoUl]). Our models of computation are *multitape Turing machines* (TM) where every tape is one-dimensional; more precisely, a TM has one read-only tape for input, several read-write tapes (the worktapes) and in case it computes a function, one write-only tape for output. A deterministic (respectively, nondeterministic) TM is abbreviated as a DTM (respectively, an NTM). Let DTIME ($T(n)$) (respectively, NTIME ($T(n)$)) denote the class of languages accepted in time $T(n)$ by a DTM (respectively, NTM).

We also use the *nondeterministic random access machines* (NRAM) defined in [Mo] and [Gr2]. An NRAM consists of a finite program that operates on a sequence of registers $R_0, R_1, R_2, \cdots$. Each register can store any natural number. The program is a finite sequence of instructions, labeled inst 0, inst 1, $\cdots$, inst l, of the following types:

(1) Read (R_i)	(7) $R_i := \langle R_j \rangle$
(2) $R_i := 0$	(8) $\langle R_i \rangle := R_j$
(3) $R_i := R_i + 1$	(9) Go to inst i_0 or i_1
(4) $R_i := R_i \dot{-} 1$	(10) If $R_i = R_j$ then go to inst i_0 else go to inst i_1
(5) Guess (R_i)	(11) Accept
(6) $R_i := R_j$	(12) Reject

$\langle R_i \rangle$ denotes the register pointed to by register R_i (i.e., the address of $\langle R_i \rangle$ is the content

of R_i). The effect of instructions (2)-(4), (6)-(8), and (10)-(12) is evident (the value of $x \dot{-} y$ is $x-y$ if $x \geqq y$; if $x<y$, it is 0). The control of the program is transferred from one instruction to the next one, except after instructions (9), (10). Instructions (5) and (9) are nondeterministic; the meaning of Guess (R_i) is the following: guess any integer to be stored in R_i.

At the beginning of the computation of an NRAM, the control of the program points to inst 0, the content of each register is 0, and a sequence of integers $w_0, w_1, \cdots, w_{m-1}, 0$ with $w_j > 0, j < m$, serve as inputs. The instruction Read (R_i) causes the NRAM to transfer the first integer w_j (or 0), which has not been read in up to this time into register R_i. We assume that the execution of any instruction only requires one time unit.

For convenience, we confuse each function (respectively, constant, relation) symbol and its interpretation.

The *finite spectrum* of a first-order sentence φ, denoted SPECTRUM (φ), is the set of cardinalities of (the universes of) its finite models [Chke]. We also define in § 5 a variant of the *generalized spectrum* (see [Fa]).

If n_1, n_2 are integers such that $n_1 < n_2$, then $[n_1, n_2[\stackrel{\text{def}}{=} \{n: n \text{ is an integer such that } n_1 \leqq n < n_2\}$. For convenience, we identify each integer $n > 0$ with the interval $[0, n[$.

Notation. For a real number $r > 0$, let $\log r$ denote the logarithm of r in base 2 and let $\lceil r \rceil$ and $\lfloor r \rfloor$ denote the least integer $n_0 \geqq r$ and the greatest integer $n_1 \leqq r$, respectively.

A *completely specified deterministic finite state automaton* (in short, a *complete automaton* or, simply, an *automaton*) is a tuple

$$A = (Q, \Sigma, \delta, q_0, F)$$

where Q is the set of *states*, Σ is the *input alphabet*, δ is a *transition function* from $Q \times \Sigma$ into Q, $q_0 \in Q$ is the *initial state*, and $F \subseteq Q$ is the set of "*accept*" *states*.

As usual, δ will also denote the standard extension of the transition function for words of Σ^*, i.e., δ is a function from $Q \times \Sigma^*$ into Q. The *language accepted* by A, denoted $L(A)$, is the set $L \subseteq \Sigma^*$ such that $w \in L$ if and only if $\delta(q_0, w) \in F$.

A state q is *reachable* if there is a word $w \in \Sigma^*$ such that $\delta(q_0, w) = q$. Two states q, q' are *equivalent* if for each word $w \in \Sigma^*$ we have $\delta(q, w) \in F$ if and only if $\delta(q', w) \in F$. It is well known that the language $L(A)$ does not change if we remove the states that are not reachable (and the transitions that involve them) and that $L(A)$ does not change (respectively, is modified) if we identify a pair of equivalent (respectively, reachable nonequivalent) states.

Let K be a positive integer. An automaton A is *K-reducible* if there is an automaton with K or fewer states that accepts $L(A)$. This is equivalent to each of the two following assertions:

(i) A has at most K distinct equivalence classes of reachable states.

(ii) the *minimal automaton* of A (see [HoUl]) has K or less states.

An *incompletely specified deterministic finite state automaton* (in short, an *incomplete automaton*) is a tuple $A = (Q, \Sigma, \delta, q_0, F)$ similar to a complete automaton except that δ is now a "partial" transition function from a subset of $Q \times \Sigma$ into Q. If we extend δ to a total transition function $\delta': Q \times \Sigma \to Q$, then the complete automaton $A' = (Q, \Sigma, \delta', q_0, F)$ is called a *completed automaton* of A.

The following lemma gives a convenient formulation of our problem R.I.S.A.

LEMMA 2.1. *Let K be a positive integer and $A = (Q, \Sigma, \delta, q_0, F)$ be an incomplete automaton whose all states are reachable. The following assertions are equivalent*:

(i) *A has a K-reducible completed automaton.*

(ii) *there is an equivalence relation, denoted* $\sim$, *on set* Q *with at most* K *classes such that*:

(ii.1) *if* $q \sim q'$ *then* $q \in F$ *if and only if* $q' \in F$;

(ii.2) *if* $q \sim q'$ *and if* $\delta(q, \sigma)$ *and* $\delta(q', \sigma)$ *are both defined* (*for some* $\sigma \in \Sigma$) *then* $\delta(q, \sigma) \sim \delta(q', \sigma)$.

The proof is easy and then we omit it.

3. A simplified idea of the proof of the main result. The intuitive ideas of our proof are explained by the following four claims. Let S be a set of positive integers that belongs to NTIME ($n \log n$), i.e., S is the set of input integers n accepted by an NTM that works in time $n \log n$.

CLAIM 3.1 (cf. § 4). $S \in \bigcup_{c \geqq 1} \text{NRAM}(cn)$, i.e., S is the set of input integers n accepted by an NRAM that works in time cn (for a constant $c \geqq 1$).

Idea of the proof. Monien [Mo] has proved the following more general result. If T is an "honest" function such that $T(n) \geqq n \log n$, then

$$\text{NTIME}(T(n)) \subseteq \bigcup_{c \geqq 1} \text{NRAM}(cT(n)/\log T(n)).$$

The idea is essentially to use the ability of an NRAM to do operations on integers in one step and to guess an integer in one step. More precisely, by first precomputing the table of all possible $\varepsilon \log T$ moves of the NTM (for a sufficiently small ε, the precomputation requires time $O(T/\log T)$) the NRAM simulates $\varepsilon \log T$ moves of the NTM in $O(1)$ steps by consulting the table only once. □

CLAIM 3.2 (cf. § 5). (i) If $S \in \bigcup_{c \geqq 1} \text{NRAM}(cn)$ then there is a fixed $c \geqq 1$ such that the set of integers $cS = \{cn: n \in S\}$ is the spectrum of a first-order sentence Φ of the form for all $y\Psi(y)$ (with the only variable y) where Ψ is quantifier-free.

(ii) Moreover, we can require that Ψ be of the form

$$\bigwedge_{i<p} \mathscr{F}_i(y) = \mathscr{G}_i(y)$$

where each $\mathscr{F}_i$, $\mathscr{G}_i$ is a word of the form $f_k \cdots f_1$ with $k \geqq 0$ and each f_j is a unary function symbol (the interpretations of some functions f_j may be predefined).

Idea of the proof. We encode the cn moves of an "accept" computation of our NRAM into a "computation structure" $\langle cn, <, \cdots \rangle$ on the universe $cn = \{0, 1, \cdots, cn-1\}$ where except the natural linear order $<$ of the universe we only have 0-ary or unary function symbols. Last we define the "computation structures" to be exactly the finite models of sentence Φ: the variable y intuitively represents each of the cn instants of the computation of the NRAM. To prove (ii) we first suppress the inequalities $\neq$ and $<$ by using predefined arithmetical functions. Then we suppress the disjunctions of Φ by enlarging the universe (the constant c may increase) and by adding new predefined functions on the enlarged universe. □

CLAIM 3.3 (cf. § 6). For each positive integer N there is an incomplete automaton $A(\Phi, N)$ such that

(i) N belongs to SPECTRUM (Φ) (i.e., Φ has a model of cardinality N) if and only if $A(\Phi, N)$ can be extended to a complete $(N+1)$-reducible automaton.

(ii) The reduction $N \mapsto A(\Phi, N)$ is computable in time $O(N \log N)$ on a DTM.

Idea of the proof.

—We construct the incomplete automaton $A(\Phi, N)$ so that its set of states Q includes the interval on integers $[-1, N[$, its initial state is -1, which is also the only "accept" state and no two states of $[-1, N[$ are equivalent. Consequently, $A(\Phi, N)$ (or any of its completed automata) is $(N+1)$-reducible if and only if each state of Q, except state -1, is equivalent to one of the N nonequivalent states $0, 1, \cdots$ or $N-1$.

—The input alphabet Σ of $A(\Phi, N)$ includes the set of function symbols f that occur in Φ.

—Each state of Q (except -1) is a "word" $\mathcal{F}(e)$ where $\mathcal{F}$ is a sequence (maybe empty) of function symbols such that $\mathcal{F}(y)$ is a term or a subterm of Φ and $e \in [0, N[$.

—The partial transition function δ of $A(\Phi, N)$ has transitions of the form $\mathcal{F}(e) \xrightarrow{f} f\mathcal{F}(e)$ for each state of the form $f\mathcal{F}(e)$; for each equality $\mathcal{F}_i(y) = \mathcal{G}_i(y)$ that occurs in Φ, δ has the transitions $\mathcal{F}_i(e) \xrightarrow{\text{Id}} \mathcal{G}_i(e)$ where Id is a new input symbol (which intuitively represents the "identity" function) and e is any state of $[0, N[$.

The reader should have the following idea. The "unrolled" form $\bigwedge_{e \in N} \bigwedge_{i<p} \mathcal{F}_i(e) = \mathcal{G}_i(e)$ of sentence Φ is satisfiable on universe $[0, N[$ if and only if the (incomplete) automaton $A(\Phi, N)$ (in fact, one of its completed automata) is $(N+1)$-reducible: intuitively, "equivalence" of states means "equality" of the corresponding terms.

Claim 3.3(ii) is explained as follows. The input alphabet Σ does not depend on N; the number of states is $O(N)$, each of length $O(\log N)$; then the length of the encoding of $A(\Phi, N)$ is $O(N \log N)$. □

CLAIM 3.4. Each problem S (on positive integers) that belongs to NTIME $(n \log n)$ is reducible to problem R.I.S.A. in time $O(n \log n)$ on a DTM.

Proof. We have $S \in \bigcup_{c \geq 1} \text{NRAM}(cn)$ by Claim 3.1 and then by Claim 3.2 there is an integer $c \geq 1$ and a sentence Φ such that

$$n \in S \quad \text{iff } cn \in \text{SPECTRUM}(\Phi). \text{ Therefore } n \in S$$

if and only if $A(\Phi, cn)$ can be extended to a $(cn+1)$-reducible automaton (by Claim 3.3(i)).

Moreover, by Claim 3.3(ii) the reduction $n \mapsto A(\Phi, cn)$ is computable in time $O(n \log n)$ on a DTM. □

Theorem 1.1 is similar to Claim 3.4 and its proof uses the same ideas (with technical complications).

Remarks. Claims 3.1 and 3.2 do not refer to problem R.I.S.A. Similar results are used in the proof that $\text{SAT}_<(\mathbb{N})$ is linearly NP-complete [Gr3].

We have refined the first-order sentence $\Phi \equiv \forall y \Psi(y)$ until it has the simple form of Claim 3.2(ii). This prepares a smooth and natural proof of Claim 3.3. We feel that working with the first-order sentence as long as possible is the right method of proof.

4. Technical results on reductions of complexity classes. In this section, it is convenient to use a one-to-one representation of positive integers: the dyadic notation; an integer $e > 0$ such that

$$e = \sum_{i=0}^{l} \alpha_i 2^i$$

where $\alpha_i \in \{1, 2\}$, is represented by the word $\alpha_l \alpha_{l-1} \cdots \alpha_1 \alpha_0$.

Let $S \subseteq \{1, 2\}^*$ be a language accepted by an NTM in time cn for a constant c. Let k be a (sufficiently large) fixed integer. We are going to exhibit a special NRAM denoted NRAM_k, which simulates the NTM in time $O(n/\log n)$.

An input w of the NRAM_k is not read one bit at a time (it would require time $n = \text{length}(w)$) but is read by blocks. More precisely, each word $w \in \{1, 2\}^n$ is divided into subwords $w_0, w_1, \cdots, w_{m-1}$ so that

(i) $w = w_0 w_1 \cdots w_{m-1}$ (concatenation),

(ii) For each $i < m-1$ length $(w_i) = h(n)$ where $h(n) = \lfloor 1/k \log n \rfloor$,

(iii) $0 < \text{length}(w_{m-1}) \leq h(n)$.

Consequently, $m = \lceil n/h(n) \rceil$ (we assume $n \geqq 2^k$ so that $h(n) \geqq 1$).

For convenience we identify a word w with its corresponding m-tuple $(w_0, w_1, \cdots, w_{m-1})$. Each word w_i is also identified with the integer it represents in dyadic notation. So, $0 < w_i < 2^{h(n)+1} \leqq 2n^{1/k}$. Each instruction Read (R_j) causes the NRAM_k to transfer the first integer w_i that has not been read in up to this time into register R_j (convention: $w_m = 0$).

We also identify each sufficiently long word w with the first-order structure $\langle m, f \rangle$ where the domain is $m = \{0, 1, \cdots, m-1\}$ and f is the function: $m \to m - \{0\}$ such that for all $i < m$, $f(i) = w_i$ (this is possible since $w_i < 2n^{1/k} \leqq m$ except for finitely many n). So, for any fixed integer k, a set of words $S \subseteq \{1, 2\}^*$ is identified with a set of structures $\langle m, f \rangle$ (except for finitely many words w of S; we can eliminate these words without changing the complexity of S).

Notation. Let NRAM_k $(T(m))$ denote the class of languages $S \subseteq \{1, 2\}^*$ accepted by an NRAM_k in time $T(m)$. (Note. m is not the length of the input word w but the number of subwords of its partition.)

The following result is Lemma 5.1 of [Gr3].

LEMMA 4.1. *If a language $S \subseteq \{1, 2\}^*$ belongs to* NTIME (n), *then there is a positive integer k such that $S \in \bigcup_{c \geqq 1} \text{NRAM}_k$ (cm).*

Proof. The precise proof is long and rather technical (see [Gr3, Lem. 5.1]). □

Remark. Lemma 4.1 is a speed up result since $m = \lceil n/h(n) \rceil = O(n/\log n)$.

Notation. Let S be a set of structures $\langle m, f \rangle$ where m is a positive integer and f is a function: $m \to m - \{0\}$. We write $S \in \text{NRAM}\ (T(m))$ to mean that S is accepted by an NRAM in time $T(m)$.

Remarks. m is the cardinality of the input structure (m is not the length of the encoding of this structure!). We still use the above convention: a Read instruction reads the first value $f(0), f(1), \cdots, f(m-1), 0$ that has not been read in up to this time.

LEMMA 4.2. *Let L be a problem such that each set S of structures $\langle m, f \rangle$ (where m is a positive integer and $f: m \to m - \{0\}$) that belongs to $\bigcup_{c \geqq 1}$ NRAM (cm) is reducible to L in time $O(m \log m)$ on a* DTM. *Then each problem S of* NTIME (n) *is reducible to L in time $O(n)$ on a* DTM.

Proof. Assume without loss of generality that $S \subseteq \{1, 2\}^*$. From Lemma 4.1 the fact that $S \in \text{NTIME}\ (n)$ implies that there is $k > 0$ such that $S \in \bigcup_{c \geqq 1} \text{NRAM}_k$ (cm).

From the hypothesis of Lemma 4.2 there is a reduction ρ from S to L, i.e., a function $w \mapsto \rho(w)$ of domain $\{1, 2\}^*$ such that for each n

$$\forall w \in \{1, 2\}^n, \quad w \in S \quad \text{iff } \rho(w) \in L;$$

moreover, ρ is computable on a DTM in time $O(m \log m)$ that is $O(n)$ since $m = \lceil n/h(n) \rceil$ and $h(n) = \lfloor 1/k \log n \rfloor$. □

5. NRAMs and first-order generalized spectra. We define a variant of "generalized spectrum" of a first-order sentence (see [Fa]).

DEFINITION. Let φ be a first-order sentence of type $\{\text{Fonc}, S_1, \cdots, S_k\} \cup \mathcal{T}$ where Fonc is a unary function symbol and $S_1, \cdots, S_k$ are function or constant or relation symbols. We give to each symbol S_i a precise interpretation in each domain $M = \{0, 1, \cdots, M-1\} \neq 0$ (for instance, S_1 is the natural linear order $<$ of M).

The *generalized spectrum* of φ (*for the specified interpretations* of $S_1, \cdots, S_k$), denoted GenSPECTRUM (φ), is the set of structures, of the form $\langle M, \text{Fonc} \rangle$ (for an integer $M > 0$) that have an expansion $\mathcal{M} = \langle M, \text{Fonc}, S_1, \cdots, S_k, G \rangle_{G \in \mathcal{T}}$ such that

(i) $S_1, \cdots, S_k$ have the specified interpretations in M,

(ii) $\mathcal{M}$ satisfies φ.

Symbols Fonc, $S_1, \cdots, S_k$ are called the *specified symbols* of φ. The other symbols of φ, i.e., symbols of $\mathcal{T}$, are called its *unspecified symbols.*

Notation. Let c be an integer >1 and let f be a function from m to $m-\{0\}$ where m is a positive integer. Let ${}^cf$ denote the function $cm \to cm$ such that

(i) ${}^cf(e) \stackrel{\text{def}}{=} f(e)$ for $e < m$,

(ii) ${}^cf(e) \stackrel{\text{def}}{=} 0$ for $m \leqq e < cm$.

The following lemma expresses that each linear time recognizable set is efficiently reducible to the generalized spectrum of a very simple sentence.

LEMMA 5.1 [Gr3]. *Let S be a set of structures $\langle m, f\rangle$ $(f: m \to m-\{0\})$ such that $S \in \bigcup_{c\geqq 1}$ NRAM (cm). Then there are a constant integer $c \geqq 1$, a type $\mathcal{T}$, and a sentence $\varphi \equiv \forall x \psi(x)$ (with the only variable x) of type $\{\text{Fonc}, <\} \cup \mathcal{T}$ such that*

(i) *ψ is quantifier-free.*

(ii) *The set of unspecified symbols $\mathcal{T}$ is a set of unary function symbols and constant symbols.*

(iii) *For each structure $\langle m, f\rangle$ $(f: m \to m-\{0\})$ we have $\langle m, f\rangle \in S$ if and only if $\langle cm, {}^cf\rangle \in$ GenSPECTRUM (φ) (where the specified symbols* Fonc *and $<$ are, respectively, interpreted by ${}^cf$ and the natural linear order of the domain cm).*

Proof. See Lemma 5.2 of [Gr3] for the proof. □

Let r, M be integers >1. For purposes of simplifying the form of our first-order sentence we now introduce a set

$$\mathcal{T}_r = \{\text{Zero}, \text{Div}_M, \text{Mod}_M, \text{Rep}^i_M \text{ (for each } i \in [0, r[), \pi_0, \pi_1, \pi_2, \text{Inf}\}$$

of specified unary function symbols with the following specified interpretations in each domain of the form $rM = \{0, 1, rM-1\}$. Let e denote any element of rM:

- Zero $(e) \stackrel{\text{def}}{=} 0$.
- $\text{Div}_M(e) \stackrel{\text{def}}{=} e$ div M and $\text{Mod}_M(e) \stackrel{\text{def}}{=} e$ mod M, i.e., $\text{Div}_M(e)$ and $\text{Mod}_M(e)$, respectively, denote the quotient and the remainder of the euclidean division of e by M.
- $\text{Rep}^i_M(e)$ is the representative of the equivalence class of e(modulo M) in the interval $[iM, (i+1)M[$, i.e., $\text{Rep}^i_M(e) \stackrel{\text{def}}{=} iM + (e \bmod M)$.
- Let $b = \lfloor \sqrt{M} \rfloor$ and let $e = e_2b^2 + e_1b + e_0$ be the representation of $e < M$ in base b $(0 \leqq e_i < b)$.
- π_0, π_1, π_2, denote the projections, i.e., $\pi_i(e) \stackrel{\text{def}}{=} e_i$ for $i = 0, 1, 2$.
- For each $e < b^2$ let us specify $\text{Inf}(e) \stackrel{\text{def}}{=} 1$ if $\pi_1(e) < \pi_0(e)$ and otherwise $\text{Inf}(e) \stackrel{\text{def}}{=} 0$. (Note. We specify neither $\text{Inf}(e)$ for $e \geqq b^2$ nor $\pi_i(e)$ for $e \geqq M$ and $i = 0, 1, 2$.)

—Note that the set of function symbols $\mathcal{T}_r$ depends on r (which will be fixed) but does not depend on M; we mention M in Div_M, Mod_M, and Rep^i_M to suggest their intuitive meaning.

LEMMA 5.2. *Let S be a set of structures $\langle m, f\rangle$ $(f: m \to m-\{0\})$ such that $S \in \bigcup_{c\geqq 1}$ NRAM (cm). Then there are integers c, $r \geqq 1$, a type $\mathcal{T}'$ and a sentence $\Phi \equiv \forall y \Psi(y)$ (where y is the only variable) of type $\mathcal{T}' \cup \{\text{Fonc}\} \cup \mathcal{T}_r$ such that*

(i) *Ψ is a conjunction of equalities of the form*

$$f_k \cdots f_2 f_1(y) = g_l \cdots g_2 g_1(y)$$

where $k \geqq 1$ and $l \geqq 0$.

(ii) *All unspecified symbols of Φ, i.e., elements of $\mathcal{T}'$ are unary function symbols.*

(iii) *For each structure $\langle m, f\rangle$ $(f: m \to m-\{0\})$ we have $\langle m, f\rangle \in S$ if and only if $\langle rcm, {}^{rc}f\rangle \in$ GenSPECTRUM (Φ) (where* Fonc *is interpreted by ${}^{rc}f$ and the other specified symbols of $\mathcal{T}_r$ are given their specified interpretation on rM where $M = cm$).*

Proof. It looks like the proof of Lemma 6.2 of [Gr3] (which means that if $S \in \bigcup_{c \geq 1} \mathrm{NRAM}(cm)$ then S is reducible to problem $\mathrm{SAT}_<(\mathbb{N})$ in time $O(m \log m)$ on a DTM). The essential difference is the following. Instead of first unrolling the first-order sentence of Lemma 5.1 (whose spectrum is studied) in its domain $M = cm$, we are now going to transform φ into a simpler first-order sentence Φ.

From Lemma 5.1, there is a sentence φ of type $\{\mathrm{Fonc}, <\} \cup \mathcal{T}$ of the following form:

$$\varphi \equiv \forall x \bigvee_{i<r} \bigwedge_{j<s} \sigma_j^i(x) *_j^i \tau_j^i(x)$$

such that

(i) $\mathcal{T}$ is a set of unary function symbols and constant symbols.

(ii) $*_j^i$ is written for "=" or "<".

(iii) σ_j^i and τ_j^i are terms (with the only variable x).

(iv) $\langle m, f \rangle \in S$ if and only if $\langle cm, {}^c f \rangle \in \mathrm{GenSPECTRUM}(\varphi)$.

Sentence φ is the disjunctive normal form of the sentence φ of Lemma 5.1 where negation has been eliminated.

We will transform sentence φ in two steps.

(1°) *Suppress the linear order and the constants.* We will use the specified functions π_0, π_1, π_2, Inf on the domain M (where we will take $M = cm$ for an integer $m > 0$). Recall $b = \lfloor \sqrt{M} \rfloor$ and that for each $e < M$ we have

$$e = \pi_2(e)b^2 + \pi_1(e)b + \pi_0(e) \quad \text{where each } \pi_i(e) < b.$$

Let us project the linear order $<$ by using its restriction to b, denoted $<_b$. Clearly for all $u, v < M$ we have that $u < v$ if and only if $\mathrm{Less}(u, v)$ holds where $\mathrm{Less}(u, v)$ denotes the formula

$$\begin{aligned}(\pi_2(u) <_b \pi_2(v)) \vee (\pi_2(u) = \pi_2(v) \wedge \pi_1(u) <_b \pi_1(v)) \\ \vee (\pi_2(u) = \pi_2(v) \wedge \pi_1(u) = \pi_1(v) \wedge \pi_0(u) <_b \pi_0(v)).\end{aligned}$$

Therefore the meaning of sentence φ does not change if each subformula of the form $\sigma(x) < \tau(x)$ is replaced by $\mathrm{Less}(\sigma(x), \tau(x))$. In the sentence φ so transformed each atomic subformula which is not an equality has the form $\sigma(x) <_b \tau(x)$. Such an inequality is equivalent to the following formula, denoted $\sigma\text{-Less}_b\text{-}\tau(x)$:

$$\exists z[\pi_2(z) = 0 \wedge \pi_1(z) = \sigma(x) \wedge \pi_0(z) = \tau(x) \wedge \mathrm{Inf}(z) = 1].$$

(Note. $\pi_2(z) = 0$ means $z < b^2$.)

- So, we replace each inequality $\sigma(x) <_b \tau(x)$ of φ by the formula $\sigma\text{-Less}_b\text{-}\tau(x)$. (Note. Now the only atomic subformulas are equalities.) We eliminate the $(\exists z)$ introduced by a standard skolemization. Let $\mathcal{T}''$ be the type $\mathcal{T}$ increased by the required Skolem function symbols. (Note that all these function symbols are unary.)

- We have succeeded in eliminating the linear order $<$ of φ (inside the original universe M). We do not eliminate the constant symbols $C \in \mathcal{T}''$ but regard them as unary function symbols by replacing each occurrence of C by $C(0)$. Then we replace each occurrence of zero by the term $\mathrm{Zero}(x)$.

(2°) *Suppress the disjunctions.* Note that the sentence φ obtained in part (1) above is of the form $\forall x \psi(x)$ where ψ is a Boolean combination of equalities without negation. Put φ into disjunctive normal form:

$$\forall x \bigvee_{i<r} \bigwedge_{j<s} \sigma_j^i(x) = \tau_j^i(x).$$

Now let us use the same ideas as in the proof of Lemma 6.2 of [Gr3]. In particular, we are going to extend the universe to $rM = \{0, 1, \cdots, rM - 1\}$ in order to suppress the disjunction $\bigvee_{i<r}$.

• The sentence Φ will be the conjunction of four sentences φ_1, φ_2, φ_3, φ_4, which must be interpreted in universe rM. (For greater readability some of them will be first given in a less formal but simpler manner.)

• Let φ_1' denote the conjunction

$$(\forall x < M) \bigwedge_G G(x) < M,$$

where G is any function symbol of $\mathcal{T}''$. φ_1' is equivalent to the following sentence φ_1, defined in universe rM:

$$\varphi_1 \equiv \forall y \bigwedge_G \mathrm{Div}_M\,(G(\mathrm{Mod}_M(y))) = \mathrm{Zero}\,(y).$$

(Note that $\mathrm{Div}_M\,(u) = 0$ if and only if $u < M$ and that each $x < M$ is of the form $\mathrm{Mod}_M\,(y)$ for some $y < rM$.)

• For the purpose of eliminating the disjunction $\bigvee_{i<r}$ we now represent each integer $x < M$ by its class modulo M in the interval $[0, rM[$: $\bar{x} \overset{\text{def}}{=} \{x, x+M, \cdots, x+(r-1)M\}$. We need for that two new unary function symbols P_0, P_1. The following sentence φ_2 means that P_0, P_1 permute each class $\bar{x}$ and are inverses of each other:

$$\varphi_2 \equiv \forall y \left[\bigwedge_{j=0,1} \mathrm{Mod}_M\,(P_j(y)) = \mathrm{Mod}_M\,(y) \wedge P_0P_1(y) = y \right].$$

• For all $i < r$ and $j < s$ let us introduce unary function symbols F_j^i and G_j^i (which intuitively represent terms σ_j^i and τ_j^i, respectively). Let $\mathcal{T}'$ be the union set $\mathcal{T}'' \cup \{P_0, P_1\} \cup \{F_j^i, G_j^i : i < r, j < s\}$

$$\varphi_3' \equiv (\forall x < M) \bigwedge_{i<r} \bigwedge_{j<s} [F_j^i(x) = \sigma_j^i(x) \wedge G_j^i(x) = \tau_j^i(x)],$$

$$\varphi_4' \equiv (\forall x < M) \bigwedge_{i<r} \bigwedge_{j<s} F_j^i(P_0(x+iM)) = G_j^i(P_0(x+iM)).$$

Because of the property φ_2 of permutation P_0, it is clear that $\varphi_3' \wedge \varphi_4'$ implies that φ holds in M, that is,

$$(\forall x < M) \bigvee_{i<r} \bigwedge_{j<s} \sigma_j^i(x) = \tau_j^i(x).$$

($\varphi_3' \wedge \varphi_4'$ implies that for each $x < M$ the conjunction $\bigwedge_{j<s} \sigma_j^i(x) = \tau_j^i(x)$ is satisfied for the index $i < r$ (depending on x) such that $P_0(x+iM) = x$.)

• *Conversely.* Assume φ holds and construct a permutation P_0 of rM and functions F_j^i, $G_j^i : rM \to M$ as follows. For each $x < M$ choose an index i such that $\bigwedge_{j<s} \sigma_j^i(x) = \tau_j^i(x)$ holds and take $P_0(x+iM) \overset{\text{def}}{=} x$; then complete the definition of P_0 so that it permutes each class $\bar{x}$ $(x < M)$ modulo M.

For all $x < M$, $i < r$ and $j < s$, take

$$F_j^i(x) \overset{\text{def}}{=} \sigma_j^i(x) \quad \text{and} \quad G_j^i(x) \overset{\text{def}}{=} \tau_j^i(x)$$

and for $1 \leqq i' < r$, take

$$F_j^i(x+i'M) \overset{\text{def}}{=} 0 \quad \text{and} \quad G_j^i(x+i'M) \overset{\text{def}}{=} 0.$$

By definition, φ_3' is trivially satisfied and so, conjuncts of φ_4' corresponding to couples (x, i) such that $P_0(x+iM) = x$ are also satisfied. The other conjuncts of φ_4' hold because

$$F_j^i(x+i'M) = G_j^i(x+i'M) \quad \text{for each } i' \in [1, r[.$$

• For the purpose of eliminating the restricted quantifiers (for all $x < M$) we now transform φ_3' and φ_4' in two sentences φ_3 and φ_4 interpreted in rM:

$$\varphi_3 \equiv \forall y \bigwedge_{i<r} \bigwedge_{j<s} [F_j^i(\mathrm{Mod}_M(y)) = \sigma_j^i(\mathrm{Mod}_M(y)) \wedge G_j^i(\mathrm{Mod}_M(y)) = \tau_j^i(\mathrm{Mod}_M(y))],$$

$$\varphi_4 \equiv \forall y \bigwedge_{i<r} \bigwedge_{j<s} F_j^i(P_0(\mathrm{Rep}_M^i(y))) = G_j^i(P_0(\mathrm{Rep}_M^i(y))).$$

(Note. Function Rep_M^i has its specified interpretation in rM.)

• From the previous discussion it must be clear that (i) each model of φ of domain M can be expanded to a model of $\Phi = \varphi_1 \wedge \varphi_2 \wedge \varphi_3 \wedge \varphi_4$ of domain rM (in which $\mathrm{Fonc}(y) = 0$ for each $y \geqq M$ and symbols or $\mathscr{T}_r$ are given their specified interpretations) and that conversely: (ii) each restriction of a model of Φ (of domain rM where symbols of $\mathscr{T}_r$ are given their standard interpretations) to the subdomain M and to the subtype $\mathscr{T} \cup \{\mathrm{Fonc}\}$ (expanded by the natural order $<$ of M) is a model of φ.

• In particular, for each structure $\langle M, \mathrm{Fonc}\rangle$ where $\mathrm{Fonc}: M \to M$ we have $\langle M, \mathrm{Fonc}\rangle \in \mathrm{GenSPECTRUM}(\varphi)$ if and only if $\langle rM, {}^r\mathrm{Fonc}\rangle \in \mathrm{GenSPECTRUM}(\Phi)$ where the generalized spectra of φ and Φ are defined with the specified interpretations of their respective specified symbols. This proves assertion (iii) of Lemma 5.2 (using Lemma 5.1(iii)). Of course assertions (i)-(ii) also hold by construction of Φ. □

Remark 5.3. If $\sigma = \tau$ and $\sigma' = \tau'$ are two distinct conjuncts of conjunction Ψ then the terms σ, σ' are always distinct.

LEMMA 5.4. *Lemma* 5.2 *and Remark* 5.3 *still hold if assertion* (i) *is replaced by the following*:

(i′) Ψ *is a conjunction of equalities of the form*

$$f_k \cdots f_2 f_1(y) = g_l \cdots g_2 g_1(y) \quad \text{where } k \geqq 1,\ l \geqq 0$$

and f_1, g_1 *are unspecified function symbols.*

Proof. Modify as follows the formula Ψ of Lemma 5.2. Let us introduce a new nonspecified function symbol I (to be included in type $\mathscr{T}'$) that intuitively represents the "identity" function. Our new formula Ψ will be the conjunction of the equality $I(y) = y$ (universally quantified) that defines I and of the original formula Ψ in which each occurrence of y is replaced by $I(y)$. Clearly, the meaning of Ψ does not change and condition (i′) is now satisfied. □

6. Generalized spectra and problem R.I.S.A. The following easy lemma will be useful in our reduction of a generalized spectrum to problem R.I.S.A.

LEMMA 6.1. *Let* $A = (Q, \Sigma, \delta, q_0, F)$ *be a complete automaton where*

—Q *includes* $N+1$ *special states denoted* $\overline{-1}, \bar{0}, \bar{1}, \cdots, \overline{N-1}$.

—*the input alphabet* Σ *includes two special symbols denoted* Suc *and* Pred.

—δ *has the transitions* $\delta(\bar{e}, \mathrm{Suc}) = \overline{e+1}$ *for each* $e \in [-1, N-1[$ *and* $\delta(\bar{e}, \mathrm{Pred}) = \overline{e-1}$ *for each* $e \in [0, N[$.

—$q_0 = \overline{-1}$ *and* $F = \{\overline{-1}\}$.

Then

(i) *The equivalence class of state* $\overline{-1}$ *only contains* $\overline{-1}$.

(ii) *If* A *is* $(N+1)$*-reducible, then the* $N+1$ *equivalence classes are exactly* $\{\overline{-1}\}$, *class of* $\bar{0}$, *class of* $\bar{1}, \cdots,$ *class of* $\overline{N-1}$.

Proof. Part (i) is implied by the fact that $\overline{-1}$ is the only "accept" state.

(ii) It is sufficient to note that each state $\bar{i}$ $(0 \leqq i < N)$ is reachable (from $q_0 = \overline{-1}$) and that if $i < j < N$ then states $\bar{i}$ and $\bar{j}$ are not equivalent:

—We have $\delta(\overline{-1}, \mathrm{Suc}^{i+1}) = \bar{i}$, so $\bar{i}$ is reachable.

—We have $\delta(\bar{i}, \mathrm{Pred}^{i+1}) = \overline{-1}$ and $\delta(\bar{j}, \mathrm{Pred}^{j+1}) = \overline{j-i-1} \neq \overline{-1}$ therefore $\bar{i}$, $\bar{j}$ are not equivalent states. □

DEFINITION 6.2. To each pair (Φ, Fonc) where $\Phi \equiv \forall y \Psi(y)$ is the sentence of type $\mathscr{T}' \cup \{\mathrm{Fonc}\} \cup \mathscr{T}_r$ of Lemma 5.4 (r is a fixed integer) and Fonc is any function: $rM \to rM$ (where M is a positive integer), associate the incomplete automaton,

$$A(\Phi, \mathrm{Fonc}) \stackrel{\mathrm{def}}{=} (Q, \Sigma, \delta, q_0, F)$$

such that

—$Q \stackrel{\mathrm{def}}{=} \{\overline{-1}\} \cup \{\overline{\mathscr{F}(e)}$: $\mathscr{F}$ is a sequence (maybe empty) of function symbols such that $\mathscr{F}(y)$ is a term or a subterm of Φ and e is an integer of $[0, rM[\}$.

—$\Sigma = \{\mathrm{Suc}, \mathrm{Pred}\} \cup \mathscr{T}_r \cup \{\mathrm{Fonc}, \mathrm{Id}\} \cup \mathscr{T}'$ where

$$\mathscr{T}_r = \{\mathrm{Zero}, \mathrm{Div}_M, \mathrm{Mod}_M, \mathrm{Rep}^i_M \text{ (for each } i \in [0, r[), \pi_0, \pi_1, \pi_2, \mathrm{Inf}\}.$$

—$q_0 \stackrel{\mathrm{def}}{=} \overline{-1}$ and $F \stackrel{\mathrm{def}}{=} \{\overline{-1}\}$.

—The partial transition function δ is given by definitions (1)-(5) below.

(1°) $\delta(\bar{e}, \mathrm{Suc}) \stackrel{\mathrm{def}}{=} \overline{e+1}$ for each $e \in [-1, rM-1[$ and $\delta(\bar{e}, \mathrm{Pred}) \stackrel{\mathrm{def}}{=} \overline{e-1}$ for each $e \in [0, rM[$.

(2°) For each specified function symbol f of Φ, i.e., each $f \in \mathscr{T}_r \cup \{\mathrm{Fonc}\}$ and each integer e of the specified domain of f: $\delta(\bar{e}, f) \stackrel{\mathrm{def}}{=} \overline{f(e)}$ where $f(e)$ denotes the integer the specified function f associates with e.

(3°) For each term or subterm of Φ of the form $f\mathscr{F}(y)$ where f is any (specified or unspecified) function symbol and $\mathscr{F}$ is any sequence (may be empty) of function symbols:

$$\delta(\overline{\mathscr{F}(e)}, f) \stackrel{\mathrm{def}}{=} \overline{f\mathscr{F}(e)} \quad \text{for each } e \in [0, rM[.$$

(4°) $\delta(\bar{e}, Id) \stackrel{\mathrm{def}}{=} \bar{e}$ for each $e \in [0, rM[$.

(5°) For each conjunct $\mathscr{F}(y) = \mathscr{G}(y)$ of Φ (where $\mathscr{F}$, $\mathscr{G}$ denote two sequences of function symbols):

$$\delta(\overline{\mathscr{F}(e)}, \mathrm{Id}) \stackrel{\mathrm{def}}{=} \overline{\mathscr{G}(e)} \quad \text{for each } e \in [0, rM[.$$

This long definition requires some comments.

Comments.

—Definitions of q_0, F, and of transitions (1) are exactly what we need to apply Lemma 6.1.

—A consequence of transitions (3) is that if $f_k \cdots f_1(y)$ is a term or a subterm of Φ then we have $\delta(\bar{e}, f_1 \cdots f_k) = \overline{f_k \cdots f_1(e)}$ for each $e \in [0, rM[$.

—Note that there is no "conflict" between transitions (2) and (3) because in each subterm of Φ of the form $f_1(y)$, f_1 is an *unspecified* symbol (by Lemma 5.4).

—From Lemma 5.4 there is no conflict between transitions of δ involving symbol Id. More precisely, (4) and (5) do not define twice the same transition (by Remark 5.3).

LEMMA 6.3. *Let $\Phi = \forall y \Psi(y)$ be the sentence (of type $\mathscr{T}' \cup \{\mathrm{Fonc}\} \cup \mathscr{T}_r$) of Lemma 5.4 and let* Fonc *denote any function $rM \to rM$.*

(1°) *Let $A = A(\Phi, \mathrm{Fonc}) = (Q, \Sigma, \delta, q_0, F)$ be the incomplete automaton associated with the pair (Φ, Fonc) by Definition 6.2. Then we have the equivalence*:

(i) *A has a completed automaton that is $(rM+1)$-reducible if and only if*

(ii) $\langle rM, \mathrm{Fonc}\rangle \in \mathrm{GenSPECTRUM}(\Phi)$.

This implies that for a fixed sentence Φ the function

$$\langle rM, \mathrm{Fonc}\rangle \mapsto (A, K)$$

where $A = A(\Phi, \text{Fonc})$ *and* $K = rM+1$ *is a reduction from* GenSPECTRUM (Φ) *to problem* R.I.S.A.

(2°) *This reduction is computable in time* $O(M \log M)$ *on a* DTM.

Proof. (i) $\rightarrow$ (ii). Let $A' = (Q, \Sigma, \delta', q_0, F)$ be a completed automaton of A that is $(rM+1)$-reducible. Then applying Lemma 6.1 to A' and $N = rM$, we conclude that for each term or subterm $\mathscr{F}(y)$ of Φ and each $e \in [0, rM[$ the state $\overline{\mathscr{F}(e)}$ is equivalent to exactly one state $\overline{e'}$ where $e' \in [0, rM[$. Let us define a first-order structure $\mathscr{M}$ of type $\mathscr{T}' \cup \{\text{Fonc}\} \cup \mathscr{T}_r$ (the type of Φ). Define the universe $|\mathscr{M}|$ of $\mathscr{M}$ to be the set of equivalence classes of Q (in A') except the class of $\overline{-1}$

$$|\mathscr{M}| \stackrel{\text{def}}{=} \{\text{class}\,(\bar{e})\colon 0 \leqq e < rM\}.$$

For each $e \in [0, rM[$, let us identify class $(\bar{e})$ with the integer e.

For each specified or unspecified function symbol $f \in \Sigma$ (symbols Suc, Pred, Id can be dropped) define: $f(\text{class}\,(\bar{e})) \stackrel{\text{def}}{=} \tilde{\delta}'(\text{class}\,(\bar{e}), f)$ where $\tilde{\delta}'$ is the transition function induced by the transition function δ' on the quotient set of Q (in other words, $\tilde{\delta}'$ is the transition function of the *minimal automaton* of A').

Now let us draw the consequences of parts (2)-(5) of the definition of the partial function δ (included in the transition function δ' of A'). Part (2) means that all the specified functions f respect their specifications (recall that Fonc is a specified function). Part (3) means that for each $e \in [0, rM[$ and each term $f_k \cdots f_1(y)$ of Φ

$$\tilde{\delta}'(\text{class}\,(\bar{e}), f_1 \cdots f_k) = \text{class}\,(\overline{f_k \cdots f_1(e)}).$$

Parts (4)-(5) together imply that for each conjunct $f_k \cdots f_1(y) = g_l \cdots g_1(y)$ of Ψ

$$\text{class}\,(\overline{f_k \cdots f_1(e)}) = \text{class}\,(\overline{g_l \cdots g_1(e)})$$

for each $e \in [0, rM[$. Therefore we have in $\mathscr{M}$

$$f_k \cdots f_1(\text{class}\,(\bar{e})) = g_l \cdots g_1(\text{class}\,(\bar{e}))$$

and then

$$\mathscr{M} \models \forall y (f_k \cdots f_1(y) = g_l \cdots g_1(y)),$$

so $\mathscr{M} \models \Phi$ and then $\langle rM, \text{Fonc}\rangle \in \text{GenSPECTRUM}\,(\Phi)$.

(ii) $\rightarrow$ (i). Assume $\langle rM, \text{Fonc}\rangle \in \text{GenSPECTRUM}\,(\Phi)$, i.e., the structure $\langle rM, \text{Fonc}\rangle$ has an expansion $\mathscr{M}$ that satisfies Φ. Assertion (i) to be proved is equivalent to condition (ii) of Lemma 2.1 for $A = A(\Phi, \text{Fonc})$ and $K = rM+1$.

So we only have to exhibit an equivalence relation $\sim$ on Q that satisfies conditions (ii.1) and (ii.2) of Lemma 2.1. For each $q \in Q$ we will define its equivalence class for $\sim$, denoted class (q) as follows:

$$\text{class}\,(\overline{-1}) \stackrel{\text{def}}{=} F = \{\overline{-1}\},$$

so condition (ii.1) is trivially satisfied. Let us define exactly rM other classes, each of which includes exactly one state $\bar{e}$ for $e \in [0, rM[$:

$$\text{class}\,(\bar{e}) = \{\overline{\mathscr{F}(e')}\colon \mathscr{F}(y) \text{ is a (sub)term of } \Phi \text{ and } e' \in [0, rM[\text{ and } \mathscr{M} \models \mathscr{F}(e') = e\}.$$

Let $\tilde{Q}$ be the quotient set of Q for $\sim$. Let us define a partial transition function $\Delta\colon \tilde{Q} \times \Sigma \rightarrow \tilde{Q}$ as follows:

(1°) $\Delta(\text{class}\,(\bar{e}), \text{Suc}) \stackrel{\text{def}}{=} \text{class}\,(\overline{e+1})$ for each $e \in [-1, rM-1[$,
$\Delta(\text{class}\,(\bar{e}), \text{Pred}) \stackrel{\text{def}}{=} \text{class}\,(\overline{e-1})$ for each $e \in [0, rM[$,

(2°) If $f \in \Sigma - \{\text{Suc}, \text{Pred}, \text{Id}\}$, i.e., f is a function symbol of Φ, and if $\mathscr{M} \models f(e) = e'$ for $e, e' \in [0, rM[$, then $\Delta(\text{class}\,(\bar{e}), f) \stackrel{\text{def}}{=} \text{class}\,(\overline{e'})$,

(3°) For each $e \in [0, rM[$, $\Delta(\text{class}\,(\bar{e})), \text{Id}) \stackrel{\text{def}}{=} \text{class}\,(\bar{e})$.

It is clear that by Definition 6.2 (parts (1)-(5)) we have for all $q, q_1 \in Q$ and $f \in \Sigma$ that $\delta(q, f) = q_1$ implies $\Delta(\text{class}(q), f) = \text{class}(q_1)$.

This proves condition (ii.2) of Lemma 2.1. So assertion (1) of Lemma 6.3 is proved.

Proof of assertion (2) is easy and is left as an exercise to the reader. □

COROLLARY 6.4. *Each set S of structures $\langle m, f\rangle$ (where m is a positive integer and $f: m \to m - \{0\}$) such that $S \in \bigcup_{c \geqq 1} \text{NRAM}(cm)$ is reducible to problem* R.I.S.A. *in time $O(m \log m)$ on a* DTM.

Proof. Lemmas 5.2 and 5.4 state that there are integers $c, r \geqq 1$ and a particular sentence Φ such that S is reducible to GenSPECTRUM (Φ), i.e., more precisely,

$$\langle m, f\rangle \in S \quad \text{iff } \langle rcm, {}^{rc}f\rangle \in \text{GenSPECTRUM}(\Phi).$$

Lemma 6.3 implies the following reduction to problem R.I.S.A.:

$$\langle rcm, {}^{rc}f\rangle \in \text{GenSPECTRUM}(\Phi) \quad \text{iff } (A, K) \in \text{R.I.S.A.}$$

where $A = A(\Phi, {}^{rc}f)$ and $K = rcm + 1$. Moreover, the reduction from S to GenSPECTRUM (Φ) is obviously computable in time $O(m \log m)$ on a DTM and the second reduction is computable on a DTM in time $O(M \log M)$ where $M = cm$ (by Lemma 6.3(2)). □

Theorem 1.1 immediately follows from Corollary 6.4 and Lemma 4.2.

7. Conclusions. In our reduction of GenSPECTRUM (Φ) to instances (A, K) of problem R.I.S.A. each (incomplete) automaton $A = A(\Phi, \text{Fonc})$ that we obtain has an input alphabet Σ that only depends on Φ. We conjecture that Theorem 1.1 still holds with $\Sigma = \{0, 1\}$.[1]

We conjecture that many other natural problems are also linearly NP-complete:

—*Satisfiability problems* in logic (comparable to $\text{SAT}_<(\mathbb{N})$ [Gr3]).

—Problems of *graph contractability* and *graph homomorphism.*

—*Sub-graph isomorphism.*

We also conjecture that all the proofs of linear NP-completeness essentially use the *generalized spectrum* of a one variable first-order sentence. We feel that this notion is a *generic tool* for proofs of linear NP-completeness exactly as Cook's problem, SAT, the satisfiability of Boolean formulas is the generic NP-complete problem. It is not surprising because the idea of Cook's proof, we think, is essentially a description of a nondeterministic computation by a *first-order sentence* in a finite universe (see [Co1], [JoSe], [Fa], [RoSc]): the reduction to SAT is obtained by *unrolling* this sentence on the universe. Our strategy in this paper (as in [Gr3]) has been to *normalize* the first-order sentence as much as possible (cf. § 5) before *unrolling* it (cf. § 6). We think that our method contributes to a better understanding of the theory of NP-completeness of natural problems.

Note added in proof. Let us slightly modify our definition of generalized spectrum. Let Φ be a first-order sentence of type $\mathcal{T} = \{f_1, \cdots, f_p, g_1, \cdots, g_q\}$ where the f_i, g_j are unary function symbols; the generalized spectrum of Φ, denoted GenSPECTRUM (Φ), is the set of structures $\langle M, f_1, \cdots, f_p\rangle$ (where M is a positive integer) that have an expansion

$$\langle M, f_1, \cdots, f_p, g_1, \cdots, g_q\rangle$$

that satisfies Φ. Our technical result, which is the "cornerstone" of the theory of linear NP-completeness, can be roughly reformulated as follows.

PROPOSITION 7.1. *For each problem S that belongs to* NTIME(n) *there is a first-order formula Φ of unary type $\mathcal{T}$ (as above), such that*

[1] Our student S. Ranaivoson has recently proved this conjecture.

(i) Φ *is of the form* $\forall y \Psi(y)$ *where* $\Psi(y)$ *is a conjunction of equalities and has only one variable.*

(ii) *S is reducible in linear time* (*on a* DTM) *to* GenSPECTRUM (Φ).

(Hint. Reformulate Lemma 5.2 with this notion of generalized spectrum: interpretate $f_1, \cdots, f_p$ by the "specified functions" from M to M, i.e., respectively, Fonc (which is ${}^{rc}f$ where $f: m \to m - \{0\}$ is the original input function and $M = rcm$) and the functions of $\mathcal{T}_r$: Zero, $\mathrm{Div}_M, \cdots$.)

REFERENCES

[ChKe] C. C. CHANG AND L. J. KEISLER, *Model Theory*, North-Holland, Amsterdam, 1973.

[Co1] S. A. COOK, *The complexity of theorem-proving procedures*, in Proc. 3rd Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1971, pp. 151-158.

[Co2] ———, *An overview of computational complexity*, Comm. ACM, 26 (1983), pp. 400-408.

[Co3] ———, *Short propositional formulas represent nondeterministic computations*, Inform. Process. Lett., (1987/88), pp. 269-270.

[De] A. K. DEWDNEY, *Linear time transformations between combinatorial problems*, Internat. J. Comput. Math., 11 (1982), pp. 91-110.

[Fa] R. FAGIN, *Generalized first-order spectra and polynomial-time recognizable sets*, in Complexity of Computations, R. Karp, ed., SIAM-AMS Proc. 7, 1974, pp. 43-73.

[GaJo] M. R. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, 1979.

[Gr1] E. GRANDJEAN, *The spectra of first-order sentence and computational complexity*, SIAM J. Comput., 13 (1984), pp. 356-373.

[Gr2] ———, *Universal quantifiers and time complexity of random access machines*, Math. Systems Theory, 18 (1985), pp. 171-187.

[Gr3] ———, *A natural* NP-*complete problem with a nontrivial lower bound*, SIAM J. Comput., 17 (1988), pp. 786-809.

[Gr4] ———, *First-Order Spectra with One Variable*, Lecture Notes in Computer Science 270, in memoriam D. Roedding, E. Boerger, ed., 1987, pp. 166-180; in J. Comput. System Sci., to appear in revised form.

[Ho] J. E. HOPCROFT, *An n* log *n algorithm for minimizing states in a finite automaton*, in Theory of machines and computations, Z. Kohavi and A. Paz, eds., Academic Press, New York, 1971, pp. 189-196.

[HoUl] J. E. HOPCROFT AND J. D. ULLMANN, *Introduction to Automata Theory, Languages and Computation*, Addison-Wesley, Reading, MA, 1979.

[JoSe] N. G. JONES AND A. L. SELMAN, *Turing machines and the spectra of first-order formulas*, J. Symbolic Logic, 39 (1974), pp. 139-150.

[Ko] Z. KOHAVI, *Switching and Finite Automata Theory*, McGraw Hill, New York, 1970.

[Ly] J. F. LYNCH, *Complexity classes and theories of finite models*, Math. Systems Theory, 15 (1982), pp. 127-144.

[Mo] B. MONIEN, *About the derivation languages of grammars and machines*, in 4th Internat. Colloq. Automata Languages Programming, 1977, pp. 337-351.

[PPST] W. J. PAUL, N. PIPPENGER, E. SZEMEREDI, AND W. T. TROTTER, *On determinism versus nondeterminism and related problems*, in Proc. 24th Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1983, pp. 429-438.

[PaUn] M. PAULL AND S. UNGER, *Minimizing the number of states in incompletely specified sequential switching functions*, IRE Trans. Electron. Comput., 8 (1959), pp. 356-367.

[Pf] C. P. PFLEEGER, *State reduction in incompletely specified finite-state machines*, IEEE Trans. Comput., 22 (1973), pp. 1099-1102.

[ReMe] B. REUSCH AND W. MERZENICH, *Minimal coverings for incompletely sequential machines*, Acta Inform., 22 (1986), pp. 663-678.

[RoSc] D. ROEDDING AND H. SCHWICHTENBERG, *Bemerkungen zum Spectralproblem*, Z. Math. Logik Grundlag. Math., 18 (1972), pp. 1-12.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 452-456, June 1990

004

MULTIPLICATION OF POLYNOMIALS OVER FINITE FIELDS*

NADER H. BSHOUTY† AND MICHAEL KAMINSKI†

Abstract. The authors prove the $2.5n - o(n)$ lower bound on the number of multiplications/divisions required to compute the coefficients of the product of two polynomials of degree n over a finite field by means of straight-line algorithms.

Key words. polynomial multiplication, bilinear and quadratic algorithms, linear recurring sequences, Hankel matrices

AMS(MOS) subject classifications. 12E05, 12E20, 68Q40

1. Introduction. The number of multiplications/divisions required for computing the product of two degree-n polynomials over an infinite field is known to be $2n+1$. The method is to evaluate both polynomials at each of $2n+1$ distinct points (allowing ∞), multiplying and interpolating the result. This method fails for the fields with the number of elements less than $2n$. The bilinear and quadratic complexity of polynomial multiplication over finite fields has been widely studied in the literature (cf. [1], [2], [6], [8]-[11]). The best known lower bounds on the bilinear complexity of multiplying two degree-n polynomials over finite fields are as follows. For the binary field the bound is $3.52n$ (cf. [2]). The same bound holds for the quadratic complexity (cf. [10] and [8]), where the proofs can be applied to quadratic algorithms as well. For the fields with more than two elements the bound is $3n - o(n)$ (cf. [9]). Lemma 1 in this paper together with the results from [9] show that the same bounds also hold for the quadratic complexity.

The proofs of the above bounds are based on a special structure of quadratic algorithms, namely, on the fact that the multiplications in quadratic algorithms are independent of each other. It is known from [13] that if a set of quadratic forms over an infinite field can be computed in t multiplications/divisions, then it can be computed in t multiplications by a quadratic algorithm whose total number of operations differs from that of the original one by a factor of a small constant. But it is unknown whether a similar result holds for finite fields. Also, no example of a set of bilinear forms with a nontrivial lower bound on the number of multiplications/divisions required for its computation is known from the literature. In this paper we prove the $2.5n - o(n)$ lower bound on the number of multiplications/divisions required for computing the product of two degree-n polynomials over a finite field by means of straight-line algorithms.

Let $\mathbf{F}_q$ denote the q-element field and let $\mu_q(n)$ denote the number of multiplications/divisions required to compute the coefficients of the product of a polynomial of degree $n-1$ and a polynomial of degree n over $\mathbf{F}_q$ by means of straight-line algorithms. A straightforward substitution argument shows that the number of multiplications/divisions required for computing the product of two polynomials of degree n exceeds $\mu_q(n)$ by at least 1. The product of polynomials of degrees $n-1$ and n is considered for a technical reason explained in the next section.

THEOREM. *For any q we have* $\mu_q(n) > 5n/2 - n/4 \lg_q n - O(n/\lg_q^2 n)$.

The rest of the paper is organized as follows. In the next section we introduce some notation and definitions, and prove the major auxiliary technical lemmas. The proof of the lower bound is presented in § 3.

* Received by the editors May 23, 1988; accepted for publication (in revised form) August 20, 1989.
† Department of Computer Science, Technion-Israel Institute of Technology, Haifa 32000, Israel.

2. Notation and auxiliary lemmas. In this section we introduce some notation and prove the major auxiliary lemmas we shall need for the proof of the theorem.

Let k be a positive integer and let $a_0, \cdots, a_{k-1}$ be given elements of a field F. A sequence $\sigma = s_0, s_1, \cdots, s_l$ of elements of F satisfying the relation

$$s_{m+k} = a_{k-1}s_{m+k-1} + a_{k-2}s_{m+k-2} + \cdots + a_0 s_m, \qquad m = 0, 1, \cdots, l-k$$

is called a (finite kth-order homogeneous) *linear recurring sequence* in F. The terms $s_0, s_1, \cdots, s_{k-1}$ are referred to as *initial values*. The polynomial

$$f(\alpha) = \alpha^k - a_{k-1}\alpha^{k-1} - a_{k-2}\alpha^{k-2} - \cdots - a_0 \in F[\alpha]$$

is called a *characteristic polynomial* of σ. Let $\varphi(\alpha)$ be a characteristic polynomial of σ of the minimal degree. If $\deg \varphi(\alpha) + \deg f(\alpha) \leqq l+1$, then $\varphi(\alpha)$ divides $f(\alpha)$ (cf. [9, Prop. 1]). Thus if $\deg \varphi(\alpha) \leqq (l+1)/2$, then $\varphi(\alpha)$ is a unique characteristic polynomial of minimal degree. It is called the *minimal polynomial* of σ and denoted by $f_\sigma(\alpha)$.

For a sequence $\sigma = \{s_0, \cdots, s_{2n-1}\}$ we define the $(n+1) \times n$ *Hankel matrix* $H(\sigma)$ by

$$\begin{pmatrix} s_0 & s_1 & \cdots & s_{n-1} \\ s_1 & s_2 & \cdots & s_n \\ \vdots & \vdots & & \vdots \\ s_n & s_{n+1} & \cdots & s_{2n-1} \end{pmatrix}.$$

Let H^i denote the $(i+1)$st row of H, $i = 0, 1, \cdots, n$. Let k be the minimal nonnegative integer such that there exist $a_0, \cdots, a_{k-1} \in F$ satisfying

$$\sum_{i=0}^{k-1} a_i H^i = H^k.$$

The existence of k is provided by the fact that H has n columns and $n+1$ rows. We define $\tilde{\sigma} = \{\tilde{s}_0, \tilde{s}_1, \cdots, \tilde{s}_{2n-1}\}$ by the recurrence

$$\tilde{s}_{i+k} = a_{k-1}\tilde{s}_{i+k-1} + a_{k-2}\tilde{s}_{i+k-2} + \cdots + a_0 \tilde{s}_i,$$

with initial values $\tilde{s}_i = s_i$, $i = 0, \cdots, k-1$.

Let $\bar{\sigma} = \sigma - \tilde{\sigma}$. We shall denote $H(\tilde{\sigma})$ and $H(\bar{\sigma}) = H - H(\tilde{\sigma})$ by $\tilde{H}$ and $\bar{H}$, respectively. Let $f_H(\alpha) = \alpha^k - \sum_{i=0}^{k-1} a_i \alpha^i$, i.e., $f_H(\alpha)$ is a characteristic polynomials of $\tilde{\sigma}$. (In fact, $f_H(\alpha) = f_{\tilde{\sigma}}(\alpha)$, since, by definition, $f_H(\alpha)$ is a characteristic polynomial of the minimal degree.)

It can be easily verified that Lemmas 1-3 of [9] stated for $(n+1) \times (n+1)$ Hankel matrices hold for $(n+1) \times n$ Hankel matrices as well. The reason for dealing with $(n+1) \times n$ Hankel matrices is that for an $(n+1) \times (n+1)$ Hankel matrix H of rank $n+1$ the polynomial $f_H(\alpha)$ is not defined. Thus that case must be treated separately, whereas dealing with $(n+1) \times n$ Hankel matrices enables a uniform treatment.

Let $\mathbf{S}$ be a finite set of $(n+1) \times n$ Hankel matrices. Define $f_{\mathbf{S}}(\alpha) = \operatorname{lcm} \{f_H(\alpha)\}_{H \in \mathbf{S}}$, where lcm is an abbreviation for "the least common multiple", $d_{\mathbf{S}} = \deg f_{\mathbf{S}}(\alpha)$ and $r_{\mathbf{S}} = \max \{\operatorname{rank} \bar{H}\}_{H \in \mathbf{S}}$.

Below $\mathbf{x} = (x_0, x_1, \cdots, x_n)^T$ and $\mathbf{y} = (y_0, y_1, \cdots, y_{n-1})^T$ denote column vectors of indeterminates. We remind the reader that a *quadratic* (respectively, *bilinear*) algorithm for computing a set of bilinear forms of $\mathbf{x}$ and $\mathbf{y}$ is a straight-line algorithm whose

nonscalar multiplications are of the shape $L * L'$, where L and L' are linear forms in $\mathbf{x}$ and $\mathbf{y}$ (respectively, L is a linear form in $\mathbf{x}$ and L' is a linear form in $\mathbf{y}$) and each bilinear form is obtained by computing a linear combination of these products.

LEMMA 1. *Let* $\mathbf{S}=\{H_1, H_2, \cdots, H_s\}$ *be a set of* $(n+1)\times n$ *Hankel matrices. Then computing the set of bilinear forms* $\{\mathbf{x}^T H_i \mathbf{y}\}_{i=1,\cdots,s}$ *by means of quadratic algorithm requires at least* $\min\{d_{\mathbf{S}}+r_{\mathbf{S}}, n+1\}$ *multiplications.*

Proof. Assume that quadratic and bilinear complexities of the set of bilinear forms $\{\mathbf{x}^T H_i \mathbf{y}\}_{i=1,\cdots,s}$ are equal to t_Q and t_B, respectively, and let $H(\mathbf{z})=\sum_{j=1}^{s} z_j H_j$ be the characteristic matrix of the above set. Let r be the row rank of $H(\mathbf{z})$. Then $t_Q \geqq (t_B+r)/2$ (cf. [7, Thm. 3.5]). Since $t_B \geqq r=\min\{d_{\mathbf{S}}+r_{\mathbf{S}}, n+1\}$ (cf. [9, Lem. 2 and 3]), we have $t_Q \geqq \min\{d_{\mathbf{S}}+r_{\mathbf{S}}, n+1\}$. □

Let V be a vector space over F and let $W \subset V$. We shall denote the linear subspace of V spanned by all the vectors from W by $[W]$.

LEMMA 2. *Let* $\mathbf{S}$ *and* $\mathbf{S}'$ *be finite sets of* $(n+1)\times n$ *Hankel matrices such that* $[\mathbf{S}]=[\mathbf{S}']$. *If* $d_{\mathbf{S}}+r_{\mathbf{S}} \leqq n$, *then* $f_{\mathbf{S}}(\alpha)=f_{\mathbf{S}'}(\alpha)$ *and* $r_{\mathbf{S}}=r_{\mathbf{S}'}$.

Proof. Let $\mathbf{S}=\{H(\sigma_i)\}_{i=1,\cdots,k}$ and let $H(\sigma)\in\mathbf{S}'$, where $\sigma=\sum_{i=1}^{k}\lambda_i\sigma_i$. It suffices to prove that $f_H(\alpha)$ divides $f_{\mathbf{S}}(\alpha)$ and rank $\bar{H} \leqq r_{\mathbf{S}}$. By Theorem 8.55 of [12, p. 425], $f_{\mathbf{S}}(\alpha)=\alpha^{d_{\mathbf{S}}}-\sum_{i=0}^{d_{\mathbf{S}}-1}\alpha_i\alpha^i$ is a characteristic polynomial of $\sum_{i=1}^{k}\lambda_i\tilde{\sigma}_i\,(=\tilde{\sigma})$. Since $d_{\mathbf{S}}+r_{\mathbf{S}} \leqq n$, the sequences σ and $\sum_{i=1}^{k}\lambda_i\tilde{\sigma}_i$ have the same first $2n-r_{\mathbf{S}}$ elements. Therefore $H^{d_{\mathbf{S}}}=\sum_{i=0}^{d_{\mathbf{S}}-1} a_i H^i$, which implies that $f_H(\alpha)$ divides $f_{\mathbf{S}}(\alpha)$. This, in turn, implies the equality $\bar{\sigma}=\sum_{i=1}^{k}\lambda_i\bar{\sigma}_i$. Thus the first $2n-r_{\mathbf{S}}$ elements of $\bar{\sigma}$ are zero and the inequality rank $\bar{H} \leqq r_{\mathbf{S}}$ follows. □

LEMMA 3. *Let* $\mathbf{S}=\{H_1, H_2, \cdots, H_m\}$ *be a set of linearly independent* $(n+1)\times n$ *Hankel matrices such that* $d_{\mathbf{S}}+r_{\mathbf{S}} \leqq n$. *Let* l *be the number of distinct irreducible factors of* $f_{\mathbf{S}}(\alpha)$. *Then computing the set of bilinear forms* $\{\mathbf{x}^T H_i \mathbf{y}\}_{i=1,\cdots,m}$ *by means of straight-line algorithms requires at least* $m+d_{\mathbf{S}}+r_{\mathbf{S}}-l-1$ *multiplications/divisions.*

Proof. Let $F[u]$ be the ring of univariate polynomials over the field F and let $F(u)$ be the field of fractions of $F[u]$. That is, $F(u)$ is the extension of F with a transcendental element u. Then any straight-linear algorithm over F is also a straight-line algorithm over $F(u)$ and polynomials irreducible over F remain irreducible over $F(u)$. In particular, the number of irreducible factors of $f_{\mathbf{S}}(\alpha)$ over $F(u)$ is equal to l. Thus, extending F with a transcendental element, if necessary, we may assume that the field of constants F is infinite. Therefore we may restrict ourselves to quadratic algorithms (cf. [13]). Assume that all the bilinear forms defined by the matrices from $\mathbf{S}$ can be computed in t multiplications. Then there exist $2t$ linear forms $L_1(\mathbf{x},\mathbf{y}), \cdots, L_t(\mathbf{x},\mathbf{y})$ and $L'_1(\mathbf{x},\mathbf{y}), \cdots, L'_t(\mathbf{x},\mathbf{y})$ in $\mathbf{x}$ and $\mathbf{y}$ such that each $\mathbf{x}^T H_i \mathbf{y}$ is a linear combination of the products $\{L_i(\mathbf{x},\mathbf{y})L'_i(\mathbf{x},\mathbf{y})\}_{i=1,\cdots,t}$. Let $\mathbf{p}=(L_1(\mathbf{x},\mathbf{y})L'_1(\mathbf{x},\mathbf{y}), \cdots, L_t(\mathbf{x},\mathbf{y})L'_t(\mathbf{x},\mathbf{y}))^T$ and let $\mathbf{q}=(\mathbf{x}^T H_1\mathbf{y}, \cdots, \mathbf{x}^T H_m\mathbf{y})^T$. By the definition of quadratic algorithms there exists an $m\times t$ matrix U whose entries are constants from F such that $\mathbf{q}=U\mathbf{p}$. Since the matrices $\{H_i\}_{i=1,\cdots,m}$ are linearly independent, rank $U=m$.

Permuting the components of $\mathbf{p}$, if necessary, we may assume that the first m columns of U are linearly independent. Hence there exist a nonsingular $m\times m$ matrix W and an $m\times(t-m)$ matrix V such that $W\mathbf{q}=(I_m, V)\mathbf{p}$, where I_m denotes the $m\times m$ identity matrix.

Let $W\mathbf{q}=(\mathbf{x}^T H'_1\mathbf{y}, \cdots, \mathbf{x}^T H'_m\mathbf{y})$ and let $\mathbf{S}'=\{H'_1, \cdots, H'_m\}$. Since W is a nonsingular matrix, we have $[\mathbf{S}]=[\mathbf{S}']$. Therefore, by Lemma 2, $f_{\mathbf{S}}(\alpha)=f_{\mathbf{S}'}(\alpha)$ and $r_{\mathbf{S}}=r_{\mathbf{S}'}$. Let $f_{\mathbf{S}}=\prod_{i=1}^{l} f_i^{d_i}(\alpha)$ be the decomposition of $f_{\mathbf{S}}(\alpha)$ into its irreducible factors. Then there exists a sequence $H'_{j_0}, H'_{j_1}, \cdots, H'_{j_l}$ of matrices from $\mathbf{S}'$ such that rank $H'_{j_0}=r_{\mathbf{S}}$ and $f_i^{d_i}(\alpha)$ divides $f_{H'_{j_i}}(\alpha)$, $i=1, 2, \cdots, l$. Note that the matrices in the above sequence

are not necessarily distinct. Permuting the components of $\mathbf{p}$, if necessary, we may assume that $j_i \geqq m-l$, $i=0,1,\cdots,l$. Then $\operatorname{lcm}\{f_{H_{m-i}}(\alpha)\}_{i=0,1,\cdots,l}=f_{\mathbf{S}}(\alpha)$ and $\max\{\operatorname{rank} \bar{H}_{m-i}\}_{i=0,1,\cdots,l}=r_{\mathbf{S}}$. By Lemma 1, computing the bilinear forms defined by the last $l+1$ components of $W\mathbf{q}$ requires at least $d_{\mathbf{S}}+r_{\mathbf{S}}$ multiplications. Since the first $m-l-1$ components in each of the last $l+1$ rows of (I_m, V) are zero and the products $\{L_i(\mathbf{x},\mathbf{y})L_i'(\mathbf{x},\mathbf{y})\}_{i=1,\cdots,t}$ are computed independently of each other, we have $t-(m-l-1)\geqq d_{\mathbf{S}}+r_{\mathbf{S}}$. Hence $t\geqq m+d_{\mathbf{S}}+r_{\mathbf{S}}-l-1$, which completes the proof. □

3. Proof of the lower bound. We shall need the following definition. Let $M=(m_{i,j})$ be a $u\times v$ matrix. We shall say that M is in *echelon form* if there exists a $k\leqq u$ and a sequence $j_1<j_2<\cdots<j_k$ such that the folowing conditions are satisfied:

(i) All the entries in the last $u-k$ rows of M are zero.

(ii) For each $i=1,\cdots,k$ the entry m_{i,j_i} is not equal to zero.

(iii) For each $i=1,\cdots,k$ and $j<j_i$ the entry $m_{i,j}$ is zero.

It is well known that each matrix can be transformed into echelon form by a sequence of elementary operations on its rows.

Proof of the Theorem. We must compute $z_k=z_k(\mathbf{x},\mathbf{y})=\sum_{i+j=k}x_iy_j$, $k=0,\cdots,2n-1$. Let $\mathbf{z}=(z_0,z_1,\cdots,z_{2n-1})^T$. Assume that $\mu_q(n)=t$, i.e., all the bilinear forms defined by the components of $\mathbf{z}$ can be computed in t multiplications/divisions. It is known from [3] that $t\geqq 2n$. Let $m_1,m_2,\cdots,m_t$ be the outputs of the multiplications/divisions of an algorithm that computes $\{z_k\}_{k=0,\cdots,2n-1}$ such that m_i is computed prior to m_j if $i<j$. Let $\mathbf{p}=(m_t,m_{t-1},\cdots,m_1)^T$. Then there exist a $2n\times t$ matrix U whose entries are constants from $\mathbf{F}_q$ and a $2n$-dimensional column vector $\mathbf{q}$ whose components are affine forms in $\mathbf{x}$ and $\mathbf{y}$ such that $\mathbf{z}=U\mathbf{p}+\mathbf{q}$. There exists a nonsingular $2n\times 2n$ matrix W such that the matrix WU is in echelon form. Multiplying $\mathbf{z}$ by W we obtain $W\mathbf{z}=WU\mathbf{p}+W\mathbf{q}$.

Let $W\mathbf{z}=(\mathbf{x}^TH_{2n}\mathbf{y},\mathbf{x}^TH_{2n-1}\mathbf{y},\cdots,\mathbf{x}^TH_1\mathbf{y})^T$ and let $\mathbf{S}_m=\{H_1,H_2,\cdots,H_m\}$, $m=1,2,\cdots,2n$. Since the matrices $H_1,H_2,\cdots,H_{2n}$ are linearly independent, $d_{\mathbf{S}_{2n}}+r_{\mathbf{S}_{2n}}\geqq 2n$ (cf. [9, Lem. 1]). Let $\mathbf{i}_q(n)$ denote the maximal possible number of distinct factors of a polynomial of degree n over $\mathbf{F}_q$. Obviously, $\mathbf{i}_q(n)\leqq n$. Therefore there is an integer m such that $d_{\mathbf{S}_{m-1}}+r_{\mathbf{S}_{m-1}}\leqq(n+\mathbf{i}_q(n/2))/2$ and $d_{\mathbf{S}_m}+r_{\mathbf{S}_m}>(n+\mathbf{i}_q(n/2))/2$. We shall consider the cases of $d_{\mathbf{S}_m}>n$ and $d_{\mathbf{S}_m}\leqq n$ separately.

If $d_{\mathbf{S}_m}+r_{\mathbf{S}_m}>n$, then, by Lemma 1, computing the set of bilinear form defined by the last m components of $W\mathbf{z}$ requires at least $n+1$ multiplications/divisions. By the definition of echelon form, at least the first $2n-m$ components in the last m rows of WU are zero. Thus we have $t-(2n-m)\geqq n+1$. Since the matrices $H_1,H_2,\cdots,H_{m-1}$ are linearly independent, $m-1\leqq(n+\mathbf{i}_q(n/2))/2$, (cf. [9, Lem. 1]). Therefore $t\geqq(5n-\mathbf{i}_q(n/2))/2$.

If $d_{\mathbf{S}_m}+r_{\mathbf{S}_m}\leqq n$, then, by Lemma 3, computing the set of bilinear form defined by the last m components of $W\mathbf{z}$ requires at least $m+d_{\mathbf{S}_m}+r_{\mathbf{S}_m}-\mathbf{i}_q(\deg f_{\mathbf{S}_m}(\alpha))-1$ multiplications/divisions. Since at least $2n-m$ first components in the last m rows of WU are zero, we have

$$t-(2n-m)\geqq m+d_{\mathbf{S}_m}+r_{\mathbf{S}_m}-\mathbf{i}_q(\deg f_{\mathbf{S}_m}(\alpha))-1.$$

Since $d_{\mathbf{S}_m}+r_{\mathbf{S}_m}>(n+\mathbf{i}_q(n/2))/2$ and the function $n-\mathbf{i}_q(n)$ is nondecreasing, it follows that $t\geqq(5n+\mathbf{i}_q(n/2))/2-\mathbf{i}_q(n/2+\mathbf{i}_q(n/2))-1$. Obviously, $\mathbf{i}_q(n_1+n_2)\leqq\mathbf{i}_q(n_1)+\mathbf{i}_q(n_2)$. Thus in either of the cases treated above we have $t\geqq(5n-\mathbf{i}_q(n/2))/2-\mathbf{i}_q(\mathbf{i}_q(n/2))$.

It can be easily shown that $\mathbf{i}_q(n)<n/\lg_q n+O(n/\lg_2^2 n)$. In particular, the proof for $q\geqq 3$ can be found in Appendix A of [9]. Thus $\mu_q(n)=t\geqq 5n/2-n/4\lg_q n-O(n/\lg_2^2 n)$. □

REFERENCES

[1] R. W. BROCKETT AND D. DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Linear Algebra Appl., 19 (1978), pp. 207–235.

[2] M. R. BROWN AND D. P. DOBKIN, *An improved lower bound on polynomial multiplication*, IEEE Trans. Comput., 29 (1980), pp. 337–340.

[3] C. M. FEDUCCIA AND Y. ZALCSTEIN, *Algebras having linear multiplicative complexity*, J. Assoc. Comput. Mach., 24 (1977), pp. 311–331.

[4] J. HOPCROFT AND J. MUNSINSKI, *Duality applied to the complexity of matrix multiplication*, SIAM J. Comput., 2 (1973), pp. 159–173.

[5] J. JA'JA', *Optimal evaluation of pairs of bilinear forms*, SIAM J. Comput., 8 (1979), pp. 443–462.

[6] ———, *Computation of bilinear forms over finite fields*, J. Assoc. Comput. Mach., 27 (1980), pp. 822–830.

[7] ———, *On the complexity of bilinear forms with commutativity*, SIAM J. Comput., 9 (1979), pp. 713–728.

[8] M. KAMINSKI, *A lower bound for polynomial multiplication*, Theoret. Comput. Sci., 40 (1985), pp. 319–322.

[9] M. KAMINSKI AND N. H. BSHOUTY, *Multiplicative complexity of polynomial multiplication over finite fields*, J. Assoc. Comput. Mach., 36 (1989), pp. 150–170.

[10] A. LEMPEL, G. SEROUSSI, AND S. WINOGRAD, *On the complexity of multiplication in finite fields*, Theoret. Comput. Sci., 22 (1983), pp. 285–296.

[11] A. LEMPEL AND S. WINOGRAD, *A new approach to error-correcting codes*, IEEE Trans. Inform. Theory, 23 (1977), pp. 503–508.

[12] R. LIDL AND H. NIEDERREITER, *Finite Fields*, Encyclopedia of Mathematics and its Applications, Vol. 20, G.-C. Rota, ed., Addison-Wesley, Reading, MA, 1983.

[13] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184–202.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 457-466, June 1990

005

A CLASS OF PROJECTIVE TRANSFORMATIONS FOR LINEAR PROGRAMMING*

YINYU YE†

Abstract. A class of projective transformations under which potential functions are invariant for linear programming is described. As a result, a new projective algorithm converging in $O(L\sqrt{n})$ iterations is developed. The algorithm does not require centering conditions, and its convergence speed is improved by a factor $\sqrt{n}$ over Karmarkar's projective algorithm and by a constant over the recent affine potential reduction algorithm.

Key words. linear programming, projective transformations, projective algorithms, potential functions, potential reduction algorithms

AMS(MOS) subject classifications. 90C05, 90C06

1. Introduction. Since Karmarkar [8] proposed the projective algorithm for linear programming (LP), many new developments have been published in the growing literature on interior algorithms. Among those, Anstreicher [1], Gay [5], Ghellinck and Vial [6], Gonzaga [7], Todd and Burrell [12], and Ye and Kojima [15] have proposed a primal projective algorithm using dual variables. These projective algorithms (including Karmarkar's original algorithm) use a projective transformation, under which a potential function is invariant, and converge in $O(Ln)$ iterations, where L is the data length and n is the number of variables in the LP. The projective transformation used in the algorithms has a nice geometric illustration.

Recently, several efforts have been made to improve interior algorithms. Todd and Ye [13] have described a class of potential functions and proposed a primal-dual projective algorithm using a new potential function for both the primal and dual. The approach is motivated by seeking reductions in the potential function as in the ordinary projective algorithms. The algorithm converges in $O(L\sqrt{n})$ iterations, and it must start at and follow the "central path" described by Bayer and Lagarias [2], Megiddo [9], Renegar [10], and Sonnevend [11].

More recently, Ye [14] has used the class of primal-dual potential functions and has developed an affine potential algorithm that directly minimizes the potential function. The algorithm seeks reductions in a suitable potential function without using the projective transformation; it converges in $O(L\sqrt{n})$ iterations without starting at or tracing the "central path."

In this paper, we describe a class of projective transformations, under which the class of potential functions is invariant. As a result, we develop a new projective algorithm converging in $O(L\sqrt{n})$ iterations. Again, the algorithm does not require the centering condition, and its convergence speed is slightly better than that of the affine potential algorithm. We also elaborate on the difference between these two algorithms.

2. Potential functions and projective transformations. Linear programming is usually identified in the following standard forms:

$$\text{(LP)} \qquad \text{minimize} \quad c^T x,$$

$$\text{subject to} \quad x \in \Omega = \{x: Ax = b, x \geqq 0\},$$

* Received by the editors October 26, 1988; accepted for publication (in revised form) August 30, 1989.
† Department of Management Sciences, University of Iowa, Iowa City, Iowa 52242.

where c, $A \in R^{m \times n}$, and $b \in R^m$ are given, $x \in R^n$, and the superscript ^{T} denotes the transpose operation. The dual to LP can be written as follows:

(LD) maximize $b^T y$,

subject to $s = c - A^T y \geqq 0$,

where the vector $y \in R^m$ and $s \in R^n$ is called dual slacks. For all x and y that are feasible for LP and LD (Dantzig [3]), we have

$$b^T y \leqq z^* \leqq c^T x,$$

where z^* denotes the minimal (maximal) objective value of LP (LD).

We also assume the following:

(A1) The interior of both feasible regions of LP and LD are nonempty.

(A2) A has full rank.

The second assumption is merely added for simplicity.

Ye [14] used the class of potential functions

$$\psi(x, z) = \rho \ln (c^T x - z) - \sum_{j=1}^{n} \ln (x_j)$$

for the primal, and

$$\phi(x, s) = \rho \ln (x^T s) - \sum_{j=1}^{n} \ln (x_j s_j)$$

for the primal-dual, where $z \leqq z^*$ and $n \leqq \rho < \infty$. The primal potential function is used by Karmarkar [8] with $\rho = n$ for the homogeneous linear system (see also Anstreicher [1], Ghellinck and Vial [6], Gonzaga [7], and Todd and Burrell [12]), and by Gay [5] and Ye and Kojima [15] with $\rho = n+1$ for the standard linear system. The primal-dual potential function is used by Todd and Ye [13] and Ye [14] for $\rho = n + \sqrt{n}$. Freund [4] further shows that $\rho = n + \sqrt{n}$ may be the best choice to solve LP and LD via the affine potential algorithm.

The primal potential function is related to the primal-dual potential function in the following way:

$$\phi(x, s) = \psi(x, z) - \sum_{j=1}^{n} \ln (s_j) \quad \text{for } z = b^T y. \tag{1}$$

Let r be a positive integer and $x^0 \in \Omega$ be the initial interior feasible solution. Then, we describe the following class of projective transformations $T: \Omega \subset R^n \to \bar{\Omega} \subset R^{n+r}$ defined as

$$\bar{x}_{1:n} = \frac{(n+r)(X^0)^{-1}x}{r + e^T (X^0)^{-1} x},$$

$$\bar{x}_j = \frac{(n+r)}{r + e^T (X^0)^{-1} x} \quad \text{for } j = n+1, \cdots, n+r,$$

where upper-case (X) designates the diagonal matrix of the vector (x) in lower-case, $\bar{x}_{1:n}$ denotes the first n components of $\bar{x} \in R^{n+r}$, and e stands for a vector of all ones whose dimension may vary. The inverse transformation is given by

$$x = T^{-1}(\bar{x}) = \frac{X^0 \bar{x}_{1:n}}{\sum_{j=n+1}^{n+r} \bar{x}_j / r}.$$

Note that

$$\sum_{j=1}^{n+r} \bar{x}_j = n + r.$$

Thus, $\bar{\Omega}$ is contained in the $n+r$ simplex

$$\Xi = \{\bar{x} \in R^{n+r}: \bar{x} \geqq 0 \text{ and } \sum \bar{x}_j = n + r\}.$$

In particular,

$$x^0 \to e \in R^{n+r},$$

i.e., the center of Ξ.

Under this projective transformation, LP is related to the following:

(LP(z)) minimize $\bar{c}(z)^T \bar{x}$,

subject to $\bar{x} \in \bar{\Omega} = \{x \in R^{n+r}: \bar{A}x = 0, e^T x = n + r, x \geqq 0\}$,

where

$$\bar{c}(z)^T = (c^T X^0, \overbrace{-z/r, \cdots, -z/r}^{r})$$

and

$$\bar{A} = (AX^0, \overbrace{-b/r, \cdots, -b/r}^{r}).$$

Let us call $\bar{\Omega}$ with the following additional restrictions

$$\bar{x}_{n+1} = \bar{x}_{n+2} = \cdots = \bar{x}_{n+r},$$

the restricted $\bar{\Omega}$. Then, the primal potential function with $\rho = n + r$ is invariant in the restricted $\bar{\Omega}$. In other words,

$$\psi(x, z) - \psi(x^0, z) = \bar{\psi}(\bar{x}, z) - \bar{\psi}(e, z), \tag{2}$$

where

$$\bar{\psi}(\bar{x}, z) = (n + r) \ln (\bar{c}(z)^T \bar{x}) - \sum_{j=1}^{n+r} \ln \bar{x}_j$$

or

$$\bar{\psi}(\bar{x}, z) = (n + r) \ln (\bar{c}(z)^T \bar{x}) - \sum_{j=1}^{n} \ln \bar{x}_j - r \ln \bar{x}_{n+1}.$$

3. The potential reduction theorem. Let us restate the following lemma that is similar to Karmarkar [8] and its proof can be found in Ye [14].

LEMMA 1. *Let* $\bar{x} \in R^n$ *and* $\|\bar{x} - e\|_\infty < 1$. *Then*

$$\sum_{j=1}^{n} \ln \bar{x}_j \geqq (e^T \bar{x} - n) - \frac{\|\bar{x} - e\|^2}{2(1 - \|\bar{x} - e\|_\infty)},$$

where $\|\cdot\|$ (*without subscript*) *denotes the* l_2 *norm.*

Replacing n with $n + r$ in Lemma 1, and noting the simplex constraint in $\bar{\Omega}$ and $\|\cdot\|_\infty \leqq \|\cdot\|$, we have

$$\bar{\psi}(\bar{x}, z) - \bar{\psi}(e, z) \leqq (n + r) \ln \left(\frac{\bar{c}(z)^T \bar{x}}{\bar{c}(z)^T e} \right) + \frac{\|\bar{x} - e\|^2}{2(1 - \|\bar{x} - e\|)}. \tag{3}$$

Let $z^0 = b^T y^0$ for some $s^0 = c - A^T y^0 > 0$. Then, we solve the following ball constrained problem (BP(z^0)):

$$\begin{aligned} (\mathrm{BP}(z)) \qquad & \text{minimize} \quad \bar{c}(z)^T \bar{x}, \\ & \text{subject to} \quad \bar{A}\bar{x} = 0, \\ & \qquad\qquad e^T \bar{x} = n + r, \\ & \qquad\qquad \|\bar{x} - e\|^2 \leqq \beta^2, \end{aligned}$$

for some constant $0 < \beta < 1$, and denote by $\bar{x}(z)$ the minimal solution for BP(z). Thus, we have

$$\bar{x}(z) = e - \beta \frac{p(z)}{\|p(z)\|}, \tag{4}$$

where

$$p(z) = \bar{P}\bar{c}(z)$$

and $\bar{P}$ is the projection matrix onto the null space of $\bar{A}x = 0$ and $e^T x = 0$. Thus,

$$\bar{c}(z)^T(\bar{x}(z) - e) = -\beta \|p(z)\|.$$

Hence, due to (3) the reduction of the potential function is

$$\bar{\psi}(\bar{x}(z), z) - \bar{\psi}(e, z) \leqq -(n+r)\beta \frac{\|p(z)\|}{\bar{c}(z)^T e} + \frac{\beta^2}{2(1-\beta)}. \tag{5}$$

From (5), if $\|p(z)\| \geqq \alpha \bar{c}(z)^T e/(n+r)$ for some constant α, then we can select an appropriate β to reduce $\bar{\psi}$ by a constant.

Now, we focus on the expression of $p(z)$, which can be rewritten as

$$p(z) = \bar{c}(z) - \bar{A}^T \bar{y}(z) - \frac{\bar{c}(z)^T e}{n+r} e$$

where

$$\bar{y}(z) = (\bar{A}\bar{A}^T)^{-1} \bar{A}\bar{c}(z). \tag{6}$$

In terms of the coefficients and variables of LP and LD, $p(z)$ can be expressed by

$$p(z) = \begin{pmatrix} X^0(c - A^T \bar{y}(z)) \\ (b^T \bar{y}(z) - z)/r \\ \vdots \\ (b^T \bar{y}(z) - z)/r \end{pmatrix} - \frac{c^T x^0 - z}{n+r} e, \tag{7}$$

since $\bar{c}(z)^T e = c^T x^0 - z$. Regarding $\|p(z)\|$, we have the following lemma (note that the dimension of e varies from n to $n+r$).

LEMMA 2. *If*

$$\|p(z)\| \leqq \alpha \frac{\Delta}{n+r} \quad \textit{for some } 0 < \alpha < 1,$$

then

$$\bar{s}(z) \overset{\text{def}}{=} c - A^T \bar{y}(z) > 0 \quad \textit{and} \quad b^T \bar{y}(z) > z, \tag{8}$$

$$\left\| X^0 \bar{s}(z) - \frac{\bar{\Delta}}{n} e \right\| \leqq \frac{\bar{\Delta}}{n} \alpha \sqrt{\frac{n+n^2/r}{n+n^2/r-\alpha^2}}, \tag{9}$$

and

$$\left| \frac{n+r}{n} \frac{\bar{\Delta}}{\Delta} - 1 \right| \leqq \frac{\alpha}{\sqrt{n+n^2/r}} \tag{10}$$

where

$$\bar{\Delta} = \bar{s}(z)^T x^0 = c^T x^0 - b^T \bar{y}(z) \quad \textit{and} \quad \Delta = c^T x^0 - z.$$

Proof. Formulas (8) hold, since if there exists j such that $\bar{s}_j(z) \leqq 0$ or $b^T \bar{y}(z) \leqq z$, then from (7)

$$\| p(z) \| \geqq \frac{c^T x^0 - z}{n+r} - x_j^0 \bar{s}_j(z) \geqq \frac{c^T x^0 - z}{n+r} = \frac{\Delta}{n+r}$$

or

$$\| p(z) \| \geqq \frac{c^T x^0 - z}{n+r} - \frac{b^T \bar{y}(z) - z}{r} \geqq \frac{c^T x^0 - z}{n+r} = \frac{\Delta}{n+r},$$

which is a contradiction. Moreover, since $e \perp (X^0 \bar{s}(z) - (\bar{\Delta}/n)e)$,

$$\begin{aligned} \| p(z) \|^2 &= \left\| X^0 \bar{s}(z) - \frac{\Delta}{n+r} e \right\|^2 + r \left(\frac{\Delta - \bar{\Delta}}{r} - \frac{\Delta}{n+r} \right)^2 \\ &= \left\| X^0 \bar{s}(z) - \frac{\bar{\Delta}}{n} e \right\|^2 + n \left(\frac{\bar{\Delta}}{n} - \frac{\Delta}{n+r} \right)^2 + r \left(\frac{\Delta - \bar{\Delta}}{r} - \frac{\Delta}{n+r} \right)^2 \\ &= \left\| x^0 \bar{s}(z) - \frac{\bar{\Delta}}{n} e \right\|^2 + \left(n + \frac{n^2}{r} \right) \left(\frac{\bar{\Delta}}{n} - \frac{\Delta}{n+r} \right)^2. \end{aligned} \tag{11}$$

If (9) does not hold, i.e.,

$$\left\| X^0 \bar{s}(z) - \frac{\bar{\Delta}}{n} e \right\| > \frac{\bar{\Delta}}{n} \gamma$$

where

$$\gamma = \alpha \sqrt{\frac{n+n^2/r}{n+n^2/r-\alpha^2}}, \tag{12}$$

then from (11)

$$\begin{aligned} \| p(z) \|^2 &> \left(\frac{\bar{\Delta}}{n} \right)^2 \gamma^2 + \left(n + \frac{n^2}{r} \right) \left(\frac{\bar{\Delta}}{n} - \frac{\Delta}{n+r} \right)^2 \\ &\geqq \frac{(n+n^2/r)\gamma^2}{n^2/r+n+\gamma^2} \left(\frac{\Delta}{n+r} \right)^2 \\ &= \alpha^2 \left(\frac{\Delta}{n+r} \right)^2, \end{aligned} \tag{13}$$

where the second inequality is true since the quadratic function of $\bar{\Delta}/n$ yields minimum at

$$\frac{\bar{\Delta}}{n} = \frac{(n+n^2/r)}{n^2/r+n+\gamma^2} \left(\frac{\Delta}{n+r} \right).$$

Formula (13) is also a contradiction; therefore (9) must hold. Again, from (11), we have

$$\left(n+\frac{n^2}{r}\right)\left(\frac{\bar{\Delta}}{n}-\frac{\Delta}{n+r}\right)^2 \leqq \alpha^2\left(\frac{\Delta}{n+r}\right)^2,$$

which implies (10). □

Based on the above lemmas, we have the following potential reduction theorem.

THEOREM 1. *Let* $n \geqq 2$, $r=[\sqrt{n}]+1$, $\alpha=0.55$, $\beta=0.3$, *and let* x^0 *and* y^0 *be any interior feasible solutions for* LP *and* LD. *Denote* $s^0=c-A^Ty^0$, $z^0=b^Ty^0$, $x^1=T^{-1}(\bar{x}(z^0))$ *of* (4), $y^1=\bar{y}(z^0)$ *of* (6), *and* $s^1=\bar{s}(z^0)$ *of* (8). *Then, either*

$$\phi(x^1, s^0) \leqq \phi(x^0, s^0)-\delta$$

or

$$\phi(x^0, s^1) \leqq \phi(x^0, s^0)-\delta$$

where $\delta > 0.1$.

Proof. If

$$\|p(z^0)\| \geqq \alpha\frac{\Delta}{n+r}=\alpha\frac{\bar{c}(z^0)^Te}{n+r},$$

then from (1), (2), and (5) and noting

$$\bar{x}_{n+1}(z^0)=\bar{x}_{n+2}(z^0)=\cdots=\bar{x}_{n+r}(z^0),$$

we have

$$\begin{aligned}\phi(x^1, s^0)-\phi(x^0, s^0) &= \psi(x^1, z^0)-\psi(x^0, z^0)\\ &= \bar{\psi}(\bar{x}(z^0), z^0)-\bar{\psi}(e, z^0)\\ &\leqq -\beta\alpha+\frac{\beta^2}{2(1-\beta)}.\end{aligned} \tag{14}$$

Otherwise, from (8), we have

$$s^1=c-A^T\bar{y}(z^0)=c-A^Ty^1>0 \quad \text{and} \quad b^Ty^1>z^0.$$

Applying Lemma 1 to vector $nX^0s^1/\bar{\Delta}$ and noting (9), we have

$$\begin{aligned}n\ln(x^0)^Ts^1-\sum_{j=1}^n\ln(x_j^0s_j^1) &= n\ln(n(x^0)^Ts^1/\bar{\Delta})-\sum_{j=1}^n\ln(nx_j^0s_j^1/\bar{\Delta})\\ &= n\ln n-\sum_{j=1}^n\ln(nx_j^0s_j^1/\bar{\Delta})\\ &\leqq n\ln n+\frac{\|nX^0s^1/\bar{\Delta}-e\|^2}{2(1-\|nX^0s^1/\bar{\Delta}-e\|_\infty)}\\ &\leqq n\ln n+\frac{\gamma^2}{2(1-\gamma)}\\ &\leqq n\ln(x^0)^Ts^0-\sum_{j=1}^n\ln(x_j^0s_j^0)+\frac{\gamma^2}{2(1-\gamma)},\end{aligned} \tag{15}$$

where γ is given by (12), and the last relation holds due to the arithmetic and geometric mean inequality. Furthermore, from (10)

$$\bar{\Delta} < \left(1-\frac{r}{n+r}+\frac{n}{n+r}\frac{\alpha}{\sqrt{n+n^2/r}}\right)\Delta.$$

Thus,

$$r \ln \frac{(x^0)^T s^1}{(x^0)^T s^0} = r \ln \frac{\bar{\Delta}}{\Delta} \leqq \frac{r^2}{n+r}\left(-1+\frac{\alpha}{\sqrt{r+r^2/n}}\right). \tag{16}$$

Adding (15) and (16), we have

$$\phi(x^0, s^1) - \phi(x^0, s^0) \leqq \frac{r^2}{n+r}\left(-1+\frac{\alpha}{\sqrt{r+r^2/n}}\right) + \frac{\gamma^2}{2(1-\gamma)}. \tag{17}$$

Since $r = [\sqrt{n}]+1$ and $n \geqq 2$,

$$\frac{r^2}{n+r} \geqq \frac{4}{5}, \quad r+\frac{r^2}{n} \geqq 3 \quad \text{and} \quad n+\frac{n^2}{r} \geqq 4.$$

Therefore, by choosing $\alpha = 0.55$ and $\beta = 0.3$, we have the desired result from (14) and (17). □

4. The new primal projective algorithm. Theorem 1 establishes an important fact: the primal-dual potential function can be reduced by a constant via solving BP(z) on the interior of LP and LD, no matter where x^0 and y^0 are. This implies that BP(z) can be solved repeatedly. In addition, we can perform the line search to minimize the potential function. This results in the following new primal projective algorithm.

NEW PRIMAL PROJECTIVE ALGORITHM:
Given $Ax^0 = b$, $x^0 > 0$ and $s^0 = c - A^T y^0 > 0$;
let $z^0 = b^T y^0$, $r = [\sqrt{n}]+1$, and $\alpha = 0.55$;
set $k = 0$;
while $c^T x^k - b^T y^k \geqq 2^{-L}$ **do**
begin
Replace X^0 with X^k in BP(z^k) and compute
$\bar{y}(z^k)$ of (6) and $p(z^k)$ of (7);
if $\|p(z^k)\| \geqq \alpha(c^T x^k - z^k)/(n+r)$ **then**
$\bar{x}(z^k) = e - \beta p(z^k)$ where $\beta = \text{argmin}_{\beta \geqq 0}\, \phi(e - \beta p(z^k), s^k)$;
$x^{k+1} = T^{-1}(\bar{x}(z^k))$;
$y^{k+1} = y^k$, $s^{k+1} = s^k$ and $z^{k+1} = z^k$;
else
$s^{k+1} = s(\bar{z})$ where $\bar{z} = \text{argmin}_{z \geqq z^k}\, \phi(x^k, s(z))$;
$x^{k+1} = x^k$, $y^{k+1} = \bar{y}(\bar{z})$ and $z^{k+1} = b^T y^{k+1}$;
end;
$k = k+1$;
end.

Although Theorem 1 is proved by taking the primal and dual steps alternatively, it should also work by moving them simultaneously. In other words, we can make a dual movement before the primal is "struck" or "centered." Since its progress is uniquely measured by the potential function, the algorithm does not rely on tracing the central path, which implies that no stepsize restriction is enforced or required during the iterative process. The greater the reduction of the potential function, the faster the convergence of the algorithm.

Overall, the performance of the new primal projective algorithm results from the following theorem.

THEOREM 2. *Let $\phi(x^0, s^0) \leqq O(L\sqrt{n})$. Then, the new primal projective algorithm terminates in $O(L\sqrt{n})$ iterations and each iteration uses $O(n^3)$ arithmetic operations.*

Proof. In $O(L\sqrt{n})$ iterations

$$\phi(x^k, s^k) \leqq -L([\sqrt{n}]+1).$$

Then,

$$r \ln (x^k)^T s^k \leqq -L([\sqrt{n}]+1) - \left(n \ln (x^k)^T s^k - \sum_{j=1}^{n} \ln (x_j^k s_j^k) \right)$$

$$\leqq -L([\sqrt{n}]+1) - n \ln n.$$

Therefore,

$$([\sqrt{n}]+1) \ln (x^k)^T s^k < -L([\sqrt{n}]+1);$$

i.e.,

$$c^T x^k - b^T y^k = (x^k)^T s^k < 2^{-L}. \qquad \square$$

The above projective algorithm is based on primal scaling. Similarly, we can develop a projective algorithm based on dual scaling. See Gay [5] for Karmarkar's dual projective algorithm and Ye [14] for the dual affine potential algorithm.

5. Further remarks. The condition on the initial potential value is not critical. In fact, any point close to the central path (with the primal-dual gap less than 2^L) satisfies this condition; i.e., any $m \times n$ LP problem can be transformed to an $(m+1)\times(n+1)$ LP problem with known x^0 and s^0 that satisfy the initial condition. In practice, any interior point x^0 and a *strict* lower bound $z^0 < z^*$ suffices to start the algorithm; that is, it is not necessary to know y^0.

The major computational work of the algorithm is (7). $\bar{A}$ looks like an $m \times (n+r)$ matrix; however, due to the duplication of the last r columns, (7) can be written as

$$\bar{y}(z) = \left(A(X^k)^2 A^T + \left(\frac{1}{r}\right) bb^T \right)^{-1} \left(A(X^k)^2 c + \left(\frac{z}{r}\right) b \right). \tag{18}$$

We can use a rank-one technique to solve the above equation using factors of $A(X^k)^2 A^T$. In addition, we do not need to formulate $p(z)$ as an $n+r$ vector, since the last r components are identical. Also note that if $x^{k+1} = x^k$ in the algorithm, the factors of $A(X^k)^2 A^T$ are unchanged and should be reused for the next iteration.

The potential reduction at each iteration of this new projective algorithm is at least 0.1, compared to 0.05 of the affine potential algorithm [14]. In other words, the theoretical convergence speed of the new projective algorithm is twice as fast as the affine potential algorithm. Again, we can verify that Karmarkar's lower-rank updating technique can be applied to this projective algorithm reducing the average arithmetic operations of each iteration to $O(n^{2.5})$. The overall complexity of this projective algorithm is $O(Ln^3)$ operations.

The difference between the projective and affine potential algorithms can be further elaborated as follows. In the affine potential algorithm (see Ye [14]), the moving direction for the dual is

$$(A(X^k)^2 A^T)^{-1} A X^k \left(X^k s^k - \frac{\Delta^k}{\rho} e \right),$$

where $\Delta^k = (x^k)^T s^k$, and $X^k s^k - (\Delta^k/\rho)e$ is the partial gradient vector, scaled by X^k, of the primal-dual potential function. But in the projective algorithm, the moving

direction from (18) is

$$\bar{y}(z)-y^k=\left(A(X^k)^2A^T+\left(\frac{1}{r}\right)bb^T\right)^{-1}\left(A(X^k)^2c+\left(\frac{z}{r}\right)b\right)-y^k$$

$$=(A(X^k)^2A^T)^{-1}AX^k\left(X^ks^k-\frac{e^TP_{AX^k}X^ks^k}{r+e^TP_{AX^k}e}e\right),$$

where the projection matrix

$$P_{AX^k}=X^kA^T(A(X^k)^2A^T)^{-1}AX^k.$$

Note that the quantity $\Delta^k/\rho=\Delta^k/(n+r)$ is not necessarily equal to

$$\frac{e^TP_{AX^k}X^ks^k}{r+e^TP_{AX^k}e}.$$

Therefore, the weight between the descending and centering directions seems more sophisticatedly adjusted in the projective algorithm than in the affine potential algorithm. Nevertheless, both algorithms achieve a constant reduction for the same potential function along these different directions. This observation indicates that the weight used in generating the moving direction and the weight used in the potential function can be different for potential reductions. We may add another dimensional search for ρ to generate better directions and to speed up these potential-type algorithms.

REFERENCES

[1] K. M. ANSTREICHER, *A monotonic projective algorithm for fractional linear programming*, Algorithmica, 1 (1986), pp. 483–498.

[2] D. BAYER AND J. C. LAGARIAS, *The non-linear geometry of linear programming*, I. *Affine and projective scaling trajectories*, II. *Legendre transform coordinates and central trajectories*, Tech. Reports, Bell Laboratories, NJ, 1987; Trans. Amer. Math. Soc., to appear.

[3] G. B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[4] R. M. FREUND, *Polynomial-time algorithms for linear programming based on primal scaling and projected gradients of a potential function*, Sloan School of Management, Massachusetts Institute of Technology, Cambridge, MA, 1988.

[5] D. M. GAY, *A variant of Karmarkar's linear programming algorithm for problems in standard form*, Math. Programming, 37 (1987), pp. 81–90.

[6] G. DE GHELLINCK AND J.-P. VIAL, *A polynomial Newton method for linear programming*, Algorithmica, 1 (1986), pp. 425–454.

[7] C. C. GONZAGA, *Conical projection algorithms for linear programming*, Memorandum UCB/ERL M87/11, Electronic Research Laboratory, University of California, Berkeley, CA, 1987.

[8] N. KARMARKAR, *A new polynomial-time for linear programming*, Combinatorica, 4 (1984), pp. 373–395.

[9] N. MEGIDDO, *Pathways to the optimal set in linear programming*, in Progress in Mathematical Programming, N. Megiddo, ed., Springer-Verlag, New York, 1989, pp. 131–158.

[10] J. RENEGAR, *A polynomial-time algorithm, based on Newton's method, for linear programming*, Math. Programming, 40 (1988), pp. 59–93.

[11] G. SONNEVEND, *An "analytic center" for polyhedrons and new classes of global algorithms for linear (smooth, convex) programming*, in Proc. 12th IFIP Conference on System Modeling and Optimization, Budapest, Hungary, 1985.

[12] M. J. TODD AND B. P. BURRELL, *An extension of Karmarkar' algorithm for linear programming using dual variables*, Algorithmica, 1 (1986), pp. 409–424.

[13] M. J. TODD AND Y. YE, *A centered projective algorithm for linear programming*, Tech. Report 763, School of Operations Research and Industrial Engineering, Cornell University, Ithaca, NY, 1987; Math. Oper. Res., to appear.

[14] Y. YE, *An $O(n^3L)$ potential reduction algorithm for linear programming*, Working Paper 88-13, Department of Management Sciences, The University of Iowa, Iowa City, IA, 1988; Math. Programming, to appear.

[15] Y. YE AND M. KOJIMA, *Recovering optimal dual solutions in Karmarkar's polynomial algorithm for linear programming*, Math. Programming, 39 (1987), pp. 305–317.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 467-471, June 1990

006

MAXIMAL RANK OF $m \times n \times (mn-k)$ TENSORS*

NADER H. BSHOUTY†

Abstract. It is shown that the maximal rank of $m \times n \times (mn-k)$ tensors with $k \leqq \min\{(m-1)^2/2, (n-1)^2/2\}$ is greater than $mn - 4\sqrt{2k} + O(1)$.

Key words. computation complexity, bilinear forms, tensor rank

AMS(MOS) subject classifications. 68G25, 15A57

1. Introduction. A classical problem in algebraic computational complexity is to determine the minimal number of nonscalar multiplications required to evaluate some set $\sum_{i,j} \alpha_{i,j,k} x_i y_j$, $k = 1, \cdots, p$, of bilinear forms in noncommuting variables $x_1, \cdots, x_m$ and $y_1, \cdots, y_n$ over a field F. This number is equal to the *rank* of the defining 3-tensor $(\alpha_{i,j,k}) \in F^m \otimes F^n \otimes F^p$ (cf. [S]).

An interesting problem, which does not depend on the coefficients $\alpha_{i,j,k}$, is the determination of

$$R_F(m, n, p) = \max_{T \in F^m \otimes F^n \otimes F^p} \text{rank of } T,$$

the maximal rank of tensors in $F^m \otimes F^n \otimes F^p$. This problem has been studied quite extensively in [AL1], [AS], [Gat], [J]. Atkinson and Stephens [AS] give the following general reduction of $R_F(m, n, mn-k)$ with $k \leqq \min\{m, n\}$ to $R_F(k, k, k^2-k)$:

$$R_F(m, n, mn-k) = mn - k^2 + R_F(k, k, k^2-k).$$

They conjecture the bound

$$R_F(k, k, k^2-k) \geqq k^2 - \left\lceil \frac{k}{2} \right\rceil,$$

which implies

$$R_F(m, n, mn-k) \geqq mn - \left\lceil \frac{k}{2} \right\rceil.$$

The rank of such type of tensors is discussed in [AL1], [AL2], [AS], and [Gat].

In this paper we shall give a constructive proof of Atkinson and Stephens conjecture and prove the following stronger bound. If $k \leqq \min\{(m-1)^2/2, (n-1)^2/2\}$, then

$$R_F(m, n, mn-k) \geqq mn - 4\sqrt{2k} + O(1).$$

For tensors of the size $(n, n, \alpha n^2)$, $1 > \alpha > (1+\sqrt{5})/4 = 0.809$, our bound is better than those known from the literature. (See (2) in the next section.)

Atkinson and Stephens (cf. [AS]) have proved that

$$R_F(m, n, mn-k) = mn - \left\lceil \frac{k}{2} \right\rceil,$$

for $\min\{m, n\} \geqq k$, $0 \leqq k \leqq 4$. We shall prove the identity for $\max\{m, n\} \geqq k$, $0 \leqq k \leqq 4$.

In the complex field the existence of sequences of tensors with small rank converging to a tensor of high rank has led to the concept of border rank. The border rank

* Received by the editors September 14, 1988; accepted for publication (in revised form) July 13, 1989.
† Department of Computer Science, Technion-Israel Institute of Technology, Haifa 32000, Israel.

of T is the least integer t such that for any $\varepsilon > 0$ there exists a tensor $\tilde{T}$ such that $\|\tilde{T} - T\| < \varepsilon$ and rank of $\tilde{T} = t$. This definition yields

$$\mathrm{BR}(m, n, p) = \max_{T \in F^m \otimes F^n \otimes F^p} \text{border rank of } T.$$

The border rank has been investigated in papers by Strassen [S2], Lickteig [Li], and Bini [Bi]. For tensors of the above type Bini [Bi] proved that for $k < \min\{n, n\}$

$$\mathrm{BR}(m, n, k) = mn - k.$$

2. Notation and auxiliary lemma. In this section we shall prove the major auxiliary lemma needed for the proof of our theorem. The technique we shall use in the proof is used in [B], [BD], [KB], and [W] to obtain lower bounds for the rank of tensors of certain shapes.

DEFINITION 1. For $mn \geqq k$ let

$$\Delta_F(m, n, k) = mn - R_F(m, n, mn - k).$$

It is known (cf. [AS]) that if $k \leqq \min\{m, n, r, s\}$ then

$$\Delta_F(m, n, k) = \Delta_F(r, s, k). \tag{1}$$

The best lower bound known from the literature is (cf. [Ho])

$$R_F(m, n, p) \geqq \frac{mnp}{m + n + p - 2}. \tag{2}$$

And the best lower bound for tensors in $F^m \otimes F^n \otimes F^{mn-k}$ is (cf. [Ga])

$$R_F(m, n, mn - k) \geqq mn - k + \sqrt{2k} + O(1). \tag{3}$$

LEMMA. *Let $r \leqq m$, $s \leqq n$, and $k \leqq rs$. Then*

$$\Delta_F(m, n, k) \leqq \Delta_F(r, s, k).$$

Proof. Let $\mathbf{x} = (x_1, \cdots, x_r)$ and $\mathbf{y} = (y_1, \cdots, y_s)$ be vectors of indeterminates. Let $\mathscr{A} = \{A_1, \cdots, A_t\}$ be a t-element set of $r \times s$ $(t < rs)$ matrices such that, the rank of the bilinear forms defined by $\mathscr{A}$, i.e., of $\{\mathbf{x}^T A_1 \mathbf{y}, \cdots, \mathbf{x}^T A_t \mathbf{y}\}$, is $R_F(r, s, t)$. Define $\mathscr{B} = \{B_1, \cdots, B_t, \tilde{B}_1, \cdots, \tilde{B}_{mn-rs}\}$ by

$$B_i = \begin{pmatrix} A_i & \mathbf{0}_{r \times n-s} \\ \mathbf{0}_{m-r \times s} & \mathbf{0}_{m-r, n-s} \end{pmatrix}, \qquad i = 1, \cdots, t,$$

and

$$\tilde{B}_j = \begin{pmatrix} \mathbf{0}_{r \times s} & C_j \\ D_j & E_j \end{pmatrix}, \qquad j = 1, \cdots, m, n - rs,$$

where $\mathbf{0}_{k,l}$ is the $k \times l$ zero matrix and $\{\tilde{B}_j\}_{j=1,\cdots,mn-rs}$ are linearly independent matrices.

For a set of matrices $\mathscr{C}$, let $\delta(\mathscr{C})$ denote the multiplicative complexity of the bilinear forms defined by the matrices of $\mathscr{C}$. We have

$$R_F(m, n, mn - rs + t) \geqq \delta(\mathscr{B}). \tag{4}$$

By [BD, Thm. 9] we have

$$\delta(\mathscr{B}) \geqq mn - rs + \min_{\lambda_{i,j} \in F} \delta\left\{B_1 + \sum_{i=1}^{mn-rs} \lambda_{i,1} \tilde{B}_i, \cdots, B_t + \sum_{i=1}^{mn-rs} \lambda_{i,t} \tilde{B}_i\right\}.$$

Because

$$B_k + \sum_{i=1}^{mn-rs} \lambda_{i,k} \tilde{B}_i = \begin{pmatrix} A_k & * \\ * & * \end{pmatrix},$$

it follows that

$$\delta(\mathscr{B}) \geqq mn - rs + \delta(\mathscr{A}) = mn - rs + R_F(r, s, t).$$

Now by (4), we obtain

$$R_F(m, n, mn - rs + t) \geqq mn - rs + R_F(r, s, t).$$

Let $k = rs - t$. Here $k \leqq st$ is assumed. Then the lemma follows from the definition of Δ_F. □

The corollary below is a generalization of (1).

COROLLARY. *If $k \leqq \min\{m, n\}$, then for every integer r and s we have*

$$\Delta_F(m, n, k) \leqq \Delta_F(r, s, k).$$

Proof. Let $M \geqq \max\{k, r, s\}$. By (1) and the lemma we have

$$\Delta_F(m, n, k) = \Delta_F(M, M, k) \leqq \Delta_F(r, s, k). \qquad \square$$

3. Maximal rank of tensors. In this section we prove our main results.

THEOREM 1 (Atkinson and Stephens conjecture). *Let $k \leqq \max\{m, n\}$, $m, n \geqq 2$. Then*

$$R_F(m, n, mn - k) \geqq mn - \left\lceil \frac{k}{2} \right\rceil.$$

Proof. Assume $k \leqq n$. By the lemma,

$$\Delta_F(m, n, k) \leqq \Delta_F(2, k, k) = 2k - R_F(2, k, k).$$

Since

$$R_F(2, k, k) = k + \left\lfloor \frac{k}{2} \right\rfloor,$$

(cf. [J, Thm. 3.5]), we have

$$R_F(m, n, mn - k) \geqq mn - \left\lceil \frac{k}{2} \right\rceil. \qquad \square$$

Remark. According to the proof of the lemma, we can construct a tensor in $F^m \otimes F^n \otimes F^{mn-k}$ of rank $mn - \lceil k/2 \rceil$ as follows: Let $\mathscr{A} = \{A_1, \cdots, A_k\}$ be a set of k, $2 \times k$ matrices satisfying $\delta(\mathscr{A}) = k + \lfloor k/2 \rfloor$ (cf. [J, Thm. 3.5]). Let

$$\mathscr{B} = \{B_1, \cdots, B_k\} \cup \{E_{i,j}\}_{\substack{i=3,\cdots,m \\ j=1,\cdots,n}} \cup \{E_{i,j}\}_{\substack{i=1,2 \\ j=k+1,\cdots,n}},$$

where

$$B_i = \begin{pmatrix} A_i & \mathbf{0}_{k,n-k} \\ \mathbf{0}_{m-2,k} & \mathbf{0}_{m-2,n-k} \end{pmatrix},$$

and $E_{i,j}$ is the matrix with 1 in entry (i, j) and zero in all other entries. Then, as in the proof of the lemma, we have

$$\delta(\mathscr{B}) = mn - \left\lceil \frac{k}{2} \right\rceil.$$

THEOREM 2. *If* $k \leqq \min\{(m-1)^2/2, (n-1)^2/2\}$, *then*

$$R_F(m, n, mn-k) \geqq mn - 4\sqrt{2k} + O(1).$$

Proof. Let $q = \lceil \sqrt{k/2} \rceil$. Since $k \leqq \min\{(n-1)^2/2, (m-1)^2/2\}$, we have

$$2q = 2\left\lceil \sqrt{\frac{k}{2}} \right\rceil \leqq 2\left\lceil \frac{(\min\{m, n\}-1)}{2} \right\rceil \leqq \min\{n, m\}.$$

Therefore, by the lemma, we have

$$\Delta_F(m, n, k) \leqq \Delta_F(m, n, 2q^2) \leqq \Delta_F(2q, 2q, 2q^2).$$

Formula (2) implies

$$\Delta_F(2q, 2q, 2q^2) = 4q^2 - R_F(2q, 2q, 2q^2) \leqq 4q^2 - \frac{8q^4}{2q^2+4q-2}$$

$$= 8q - 20 + \frac{48q-20}{q^2+2q-1} \leqq 4\sqrt{2k} + O(1). \qquad \square$$

For tensors in $F^n \otimes F^n \otimes F^{\alpha n^2}$, $\alpha < 1$, bound (2) implies

$$R_F(n, n, \alpha n^2) \geqq n^2 - \frac{2}{\alpha} n + O(1),$$

whereas Theorem 2 gives the lower bound

$$R_F(n, n, \alpha n^2) \geqq n^2 - 4\sqrt{2(1-\alpha)}n + O(1).$$

A simple calculation shows that this bound improves the previous lower bound in the case when $\alpha > (1+\sqrt{5})/2 = 0.809$.

THEOREM 3. *Let* $0 \leqq k \leqq 4$, $\max(m, n) \geqq k$. *Then*

$$R_F(m, n, mn-k) = mn - \left\lceil \frac{k}{2} \right\rceil.$$

Proof. Let $m \geqq n$. By the corollary we have

$$\Delta_F(m, n, k) \geqq \Delta_F(m, m, k),$$

and by [AS, Thm. 2], for $0 \leqq k \leqq 4$,

$$\Delta_F(m, m, k) = \left\lceil \frac{k}{2} \right\rceil.$$

Therefore

$$R_F(m, n, mn-k) \leqq mn - \left\lceil \frac{k}{2} \right\rceil.$$

Now, the result follows from Theorem 1. $\square$

REFERENCES

[AL1] M. D. ATKINSON AND S. LLOYD, *Bounds on the ranks of some 3-tensors*, Linear Algebra Appl. 31 (1980), pp. 19-31.

[AL2] ———, *The rank of $m \times n \times (mn-2)$ tensors*, SIAM J. Comput., 12 (1983), pp. 611-615.

[AS] M. D. ATKINSON AND N. M. STEPHENS, *On the maximal multiplicative complexity of a family of bilinear forms*, Linear Algebra Appl., 27 (1979), pp. 1-8.

[B] N. H. BSHOUTY, *A lower bound for matrix multiplication*, in Proc. 29th Annual Symposium on Foundations of Computer Science, 1988, to appear.

[BD] R. W. BROCKETT AND D. DOBKIN, *On the optimal evaluation of a set of bilinear forms*, Linear Algebra Appl., 19 (1978), pp. 207-235.

[Bi] D. BINI, *Border rank of $m \times n \times (mn-q)$ tensors*, Linear Algebra Appl., 79 (1986), pp. 45-51.

[Gat] J. VON ZUR GATHEN, *Maximal bilinear complexity and codes*, University of Toronto, Toronto, Ontario, 1988, preprint.

[Gri] D. YU. GRIGORIEV, *Multiplicative complexity of a pair of bilinear forms and of polynomial multiplication*, Lecture Notes in Computer Science 46, Springer-Verlag, Berlin, New York, 1978, pp. 250-256.

[Ho] T. D. HOWELL, *Global properties of tensor rank*, Linear Algebra Appl., 22 (1978), pp. 9-23.

[J] J. JA'JA', *Optimal evaluation of pairs of bilinear forms*, SIAM J. Comput., 8 (1979), pp. 443-462.

[KB] M. KAMINSKI AND N. H. BSHOUTY, *Multiplicative complexity of polynomial multiplication over finite field*, in Proc. 28th Annual Symposium on Foundations of Computer Science, 1987, J. Assoc. Comput. Mach., to appear.

[Li] T LICKTEIG, *Typical tensorial rank*, Linear Algebra Appl., 69 (1985), pp. 95-120.

[S] V. STRASSEN, *Vermeidung von Divisionen*, J. Reine Angew. Math., 264 (1973), pp. 184-202.

[S2] ———, *Rank and optimal computation of generic tensors*, Linear Algebra Appl., 52 (1983), pp. 645-685.

[W] S. WINOGRAD, *Some bilinear forms whose multiplicative complexity depends on the field constants*, Math. Systems Theory, 10 (1976/77), pp. 169-180.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 472–499, June 1990

FLIPPING PERSUASIVELY IN CONSTANT TIME*

CYNTHIA DWORK†, DAVID SHMOYS‡, AND LARRY STOCKMEYER†

Abstract. A persuasive coin is a sufficiently unbiased source of randomness visible to sufficiently many processors in a distributed system. An algorithm is described for achieving a persuasive coin in the presence of an extremely powerful adversary where the number of rounds of message exchange among the processors is constant, independent of the number n of processors in the system as well as the number of faults, provided the total number of faulty processors does not exceed a certain constant multiple of $n/\log n$. As a corollary an $\Omega(n/\log n)$-resilient probabilistic protocol for Byzantine agreement running in constant expected time is obtained. Combining this with a generalization of a technique of Bracha, a probabilistic Byzantine agreement protocol tolerant of almost $n/4$ failures with $O(\log\log n)$ expected running time is obtained.

Key words. Byzantine agreement, distributed coin, distributed computing, fault tolerance, cryptographic protocol, probabilistic algorithm

AMS(MOS) subject classifications. 68M10, 68P25, 68Q

1. Introduction.

The problem of distributively flipping a fair coin visible to all processors in the system, in the presence of a malicious adversary, has recently received considerable attention ([1], [3], [4], [6], [7], [9]; see also [8] for a survey). Ideally, God flips an unbiased coin and the result is immediately visible to all processors in the system. In the absence of God and of faulty processors, it suffices for one processor to flip a fair coin and to broadcast the result to the other processors in the system. However, if the flipping processor is faulty it could misbehave in several ways, by flipping a biased coin, or failing to send the message properly, or even sending contradictory messages to different processors. As in a recent paper of Chor, Merritt, and Shmoys [10], we relax the requirement that *all* processors see the same value of the coin. We say a flip of a coin is *persuasive* if a sufficiently large majority of the processors see the same value, in our case, $\lfloor n/2 \rfloor + t + 1$ where n is the number of processors and t, called the *resiliency* of the protocol, is an upper bound on the number of faulty processors to be tolerated. We will exhibit and prove correct a protocol for obtaining a persuasive flip of a coin with constant bias in $O(1)$ time, where *time* is taken to be the number of rounds of message exchange. The protocol tolerates $\Omega(n/\log n)$ faulty processors. More specifically, our protocol tolerates an adversary that at each round r of communication can examine the messages to be sent by all processors and then select which processors to make faulty, based on all messages of rounds 1 through r. Having chosen the faulty processors the adversary may then choose the round r messages of the faulty processors, substituting these bad messages for the ones originally chosen by these processors. This adversary, called the *blocker* because it blocks and changes messages sent by the processors it chooses to make faulty, is the most powerful considered in the literature for any model in which the processors'

* Received by the editors November 23, 1987; accepted for publication (in revised form) March 24, 1989. A preliminary and abridged version of this paper appeared in the Proceedings of the 27th Annual IEEE Symposium on Foundations of Computer Science, Toronto, Ontario, Canada, 1986.

† IBM Research Division, K53/802, 650 Harry Road, San Jose, California 95120.

‡ School of Operations Research and Industrial Engineering, Cornell University, Ithaca, New York 14853. This work was performed while the author was with the Mathematics Department, Massachusetts Institute of Technology, Cambridge, Massachusetts, and while visiting the Mathematical Sciences Research Institute, Berkeley, California. It was supported by the National Science Foundation under grant MCS-8120790, by the Department of the Army under grant DAAG 29-85-K-0138, and by the Air Force Office of Scientific Research under grant AFOSR-86-0078.

internal states are not known to the other participants or the adversary. Since we will be using cryptography in protocols which are resilient to the blocker, we assume that the computations of the blocking adversary can be done by a probabilistic polynomial-time Turing machine. We will say more about adversaries in § 2.

The problem of achieving a *global* coin, visible to all correct processors, received attention as a result of the work of Rabin on Byzantine agreement [24]. In the *Byzantine agreement problem* [23], each processor p_i begins the protocol with an initial binary value u_i. Every correct processor must terminate with a decision value such that (i) all correct processors decide on the same value, and (ii) if all initial values are the same, say b, then all correct processors must decide b. Rabin showed that, given a sufficiently unbiased global coin, Byzantine agreement can be achieved in $O(f(n))$ expected time, where $f(n)$ is the time required to flip the coin in a system of n processors. The expected time to reach agreement increases with the bias of the coin. Rabin's result introduced the possibility of beating the lower bound of $t+1$ rounds to reach agreement by a deterministic protocol [11], [13], [20], where t is an upper bound on the number of faulty processors, should it only be possible to quickly flip a relatively fair coin. Rabin also showed how to use a global coin to obtain an asynchronous agreement protocol.

At the same time Ben-Or [2] obtained an asynchronous randomized agreement protocol, thus beating the impossibility result for agreement in a deterministic asynchronous environment [21]. Although the idea of the global coin is not explicitly spelled out in Ben-Or's paper, in effect his solution uses a persuasive coin with a large bias. When the algorithm is run in the synchronous round model, this large bias forces the expected number of rounds of communication to be exponential in n whenever t, the number of faults to be tolerated, is $\Omega(n)$. However, if t is $O(n^{1/2})$ then the expected number of rounds required is $O(1)$, thus beating the time $t+1$ lower bound for deterministic protocols.

The principal contribution of our paper is the coin flipping protocol. Specifically, we show that, under certain cryptographic assumptions (as in Goldwasser and Micali [22]), a persuasive coin with constant bias can be achieved in $O(1)$ time in the presence of $\Omega(n/\log n)$ faulty processors chosen and controlled by a blocking adversary. This is proved in two parts. In § 3 we describe the protocol for a weaker adversary. In § 4 we strengthen the protocol and its proof to handle the blocker. The weaker adversary corresponds to the case where each pair of processors is connected by a private communication channel so that faulty processors cannot eavesdrop on the conversations between correct processors. Our results for this adversary are interesting in their own right, as they require neither cryptography nor assumptions limiting the computational power of the adversary. At the same time the protocol is fairly simple, requiring relatively short messages and a small amount of computation by the correct processors.

In light of the results of Rabin and Ben-Or, this persuasive coin protocol immediately implies that Byzantine agreement can be achieved in $O(1)$ expected time in the presence of $\Omega(n/\log n)$ faulty processors chosen and controlled by a blocking adversary. The best previously known resiliency for Byzantine agreement in constant expected time, due to Bracha [5], was n^c for arbitrary constant $c<1$. Our solution not only has higher resiliency, but it is also simpler and constructive because it does not use "Bracha assignments" (described in § 5).

Prior to the work reported here, two constant expected time agreement protocols had already appeared in the literature, both with resiliency linear in n (as opposed to our resiliency linear in $n/\log n$). We now compare these results to ours. The protocol of Chor, Merritt, and Shmoys [10] cannot tolerate even one malicious processor failure.

Viewed in terms of the blocking adversary, their adversary can only block, it cannot change the messages of the processors it chooses to make faulty. This corresponds to the case of failure of omission: every message sent by a faulty processor is generated according to the protocol, but it may fail to reach its destination. By contrast, our adversary can exhibit arbitrarily malicious behavior. The second constant expected time agreement algorithm is due to Feldman and Micali [17]. Although their algorithm runs in constant expected time *per agreement*, their protocol uses $\Omega(n)$ preprocessing time, during which the processors agree on various things such as a pseudorandom number generator for producing global random bits. Since our protocols use fewer than t rounds, we do not have time to run deterministic Byzantine agreements.

To achieve fault tolerance higher than $n/\log n$, we generalize a technique of Bracha [5] to show that, for any fixed $\delta > 0$, Byzantine agreement can be achieved in $O(\log \log n)$ expected time in the presence of up to $n/(4+\delta)$ faulty processors. The adversary used in this result cannot eavesdrop on the conversations of correct processors (private channels) but has unlimited computational power.

After our work appeared [14], Feldman and Micali [15], [18] devised a probabilistic Byzantine agreement algorithm that runs in constant expected time and has the optimum resiliency $\lfloor (n-1)/3 \rfloor$ in a model with private channels (with no preprocessing required). In addition, Feldman [16] has a technique that, under a cryptographic assumption, converts a distributed algorithm designed for private channels to one that runs in a model without private channels. It is likely that this technique can be applied to give a constant expected time Byzantine agreement algorithm with optimum resiliency against the blocker in a model without private channels.

2. Models and definitions. Our model of distributed computation is similar to the synchronous model used in many previous papers, for example, Bracha [5] or Dolev and Strong [13], where the computation proceeds by rounds of message exchange. We recall the basic features of the model informally. We assume a completely connected network of n processors $P = \{p_1, p_2, \cdots, p_n\}$, where each processor is modeled as a probabilistic polynomial-time machine with a source of perfectly random bits. The computation of the system proceeds in synchronous numbered rounds, beginning with round 1. In general, during each round r each processor p_i performs a computation to produce, for each other processor p_j, a message to be sent from p_i to p_j during round r. All messages are sent simultaneously and all are reliably delivered. The processors then go on to round $r+1$. As discussed below, faulty processors can deviate from this perfectly synchronous behavior.

In proving correctness of deterministic distributed algorithms that tolerate Byzantine faults, it is convenient to place no restrictions on the behavior of faulty processors. For probabilistic algorithms, the allowed behaviors must be defined more precisely since, for example, the behavior of faulty processors at round r cannot depend on the random choices made by correct processors at rounds greater than r. In this paper we consider three types of faulty behavior. We define each by defining the capabilities of an *adversary* that chooses which processors to make faulty and chooses the messages sent by faulty processors. In general, these choices can be made according to some probability distribution; they need not be deterministic. For the first two types of adversaries, we place no restriction on the computational power of the adversary (it can even be nonrecursive), although for the blocking adversary, we must restrict the adversary to be a probabilistic polynomial-time Turing machine. In defining each type of adversary, we let F_r be the set of processors that are faulty during round r. We take $F_0 = \emptyset$ and we assume that $F_r \subseteq F_{r+1}$ for all r. If an algorithm is *t-resilient* to a particular

type of adversary, then the algorithm must operate correctly for all adversaries of that type that satisfy $|F_r| \leq t$ throughout the execution of the algorithm. (Actually, this requirement can be relaxed somewhat: Executions of our agreement algorithms with constant expected running time can be divided into epochs, with a small constant number of rounds per epoch, such that the algorithm operates correctly if at most t processors are faulty in each given epoch, but different sets of t processors may be faulty in different epochs.) In the three definitions given next, we assume that the adversary for round r is given r and all messages received by processors in F_{r-1} during rounds less than r. After a new set F_r of faulty processors is chosen by the adversary for round r, the adversary is also given all messages received by processors in F_r during all rounds less than r.

(1) In the *lock-step model with erasing*, the adversary is divided into three parts. The first part produces a new set F_r of faulty processors. The second part produces the (potentially malicious) messages to be sent by processors in F_r. The first and second parts are independent of the round r messages of the correct processors. The third part, called the *eraser*, is then given the round r messages of processors in $P - F_r$, and it produces a subset of F_r's messages to be erased. The messages actually sent by processors in F_r during round r are the messages produced by the second part that are not erased by the third part. Without the eraser, this model would be the simple lock-step model where all round r messages, from both correct and faulty processors, must be sent simultaneously. We introduce the eraser because proving that the persuasive coin algorithm is correct with the eraser facilitates correctness proofs for the two stronger adversaries. Immunity to erasing is an important property of the basic coin protocol.

(2) The next type of adversary is motivated by the fact that it could be difficult in practice to enforce simultaneous sending of all messages at a given round. At round r, the faulty processors could wait and receive the round r messages sent to them by correct processors before choosing and sending their own round r messages. In previous papers, this behavior has been called *rushing*. However, if each pair of processors has a dedicated communication line, messages sent from one correct processor to another cannot be overheard by any faulty processor. Formally, in the *rushing without eavesdropping* model, the adversary is divided into two parts. The first part, as above, produces a set F_r. The second part receives the messages sent from processors in $P - F_r$ to processors in F_r during round r, and it produces the round r messages sent by the processors in F_r.

(3) Our strongest type of adversary, the *blocking* adversary, permits eavesdropping and allows the identity of faulty processors at round r to depend even on the contents of messages sent at round r by processors that were correct at the end of round $r-1$. In other words, if a processor is made faulty for the first time at round r, its round r messages can be changed by the adversary. Formally, the inputs to the adversary at round r are the set F_{r-1}, *all* messages sent by *all* processors during rounds less than r, and *all* messages that are supposed to be sent by processors in $P - F_{r-1}$ at round r. The outputs are a set F_r and the round r messages actually sent by processors in F_r (so the round r messages of processors in $F_r - F_{r-1}$ can be changed by the adversary). (This adversary is somewhat stronger than one where the adversary must first specify the processors to be faulty in round r; the adversary then gets to see all of the messages sent between the correct processors as well as from the correct processors to the faulty processors, before sending the messages for the faulty processors. It is significant to note that neither of these models allows the complete corruption of a faulty processor, in that the internal state of a faulty processor is not revealed to the adversary.)

DEFINITION. A distributed algorithm is a *t-resilient persuasive coin with probability* ρ (for a particular type of adversarial behavior) if, at termination of the algorithm, each correct processor p_i has chosen a bit b_i and, for all $b \in \{0, 1\}$, the probability that at least $\min\{\lfloor n/2 \rfloor + t + 1, n\}$ correct processors choose b is at least ρ.

The connection between persuasive coins and Byzantine agreement is provided by the following lemma implict in the work of Ben-Or [2]. The connection was made explicit by Chor, Merritt, and Shmoys [10].

LEMMA 2.1. *Fix any of the adversary types described above. Suppose that there is a t-resilient persuasive coin with probability* ρ *that always terminates within k rounds and assume that* $n > 3t$. *Then there is a t-resilient probabilistic Byzantine agreement algorithm with expected running time of* $O(k/\rho)$ *rounds.*

In the following, log denotes the base 2 logarithm and ln denotes the natural logarithm.

3. Noncryptographic algorithms with constant expected running time.

3.1. The lock-step model with erasing. First we describe the new $\Omega(n/\log n)$-resilient persuasive coin protocol in the lock-step model with erasing and prove its correctness. In later sections the basic algorithm is modified for the two stronger adversaries in such a way that the new correctness proofs reduce to the correctness argument for the lock-step model with erasing.

ALGORITHM COIN

Algorithm for processor p_i:

1. Randomly choose an integer *value* v_i uniformly between 1 and n^4 (inclusive), and choose a random *bit* b_i.
2. Broadcast (v_i, b_i) to all processors.
3. Compute the outcome of the coin:
   - 3.1. Let W be the multiset of values received from all processors (including p_i); include only values in the proper range from 1 to n^4.
   - 3.2. Let $w_1, \cdots, w_m$ be the members of W sorted in nondecreasing order, and let $w_{m+1} = w_1 + n^4$. (By including the value $w_{m+1} = w_1 + n^4$, we have essentially bent the interval $[0, n^4]$ into a circle, identifying the points 0 and n^4.)
   - 3.3. Find the j with $1 \leqq j \leqq m$ such that $w_{j+1} - w_j$ is maximized; if there are several such j's then pick the smallest.
   - 3.4. Let p_k be the processor that sent value w_j; if there is more than one such processor, let p_k be the one with smallest index k. We say that p_k is the *leader* chosen by p_i. The value of p_i's coin is the bit b_k sent by p_k to p_i, or zero if no bit was received from p_k.

THEOREM 3.1. *Fix any constant* $c < 1/6$. *In the lock-step model with erasing, there is a constant* $\rho > 0$ *such that, for all n, Algorithm* COIN *is a* $(cn/\ln n)$*-resilient persuasive coin with probability* ρ.

Proof. Let $t = \lfloor cn/\ln n \rfloor$ be the upper bound on the number of faults. In the following, we assume $n \geqq 17$, since otherwise $t < 1$ and the protocol is certainly correct if there are no faults. Since the algorithm has only one round, there is no harm in assuming that the adversary makes as many processors faulty as possible. Therefore, the adversary chooses a set F of t faulty processors and, for each p_i in F and each p_j not in F, the adversary chooses a value a_{ij} to be sent from p_i to p_j. We show that, with probability bounded above 0, at least $\lfloor n/2 \rfloor + t + 1$ processors in $P - F$ choose the same leader p_k and that p_k is not in F. This is sufficient to prove the theorem, since p_k not in F means that p_k chooses its bit b_k randomly.

Let $V=\{v_1, \cdots, v_{n-t}\}$ be the multiset of values chosen randomly by the correct processors, sorted in nondecreasing order, and let $v_{n-t+1}=v_1+n^4$. The *size* of the maximum gap of V is the maximum of $v_{j+1}-v_j$ over $1 \leqq j \leqq n-t$. Any pair (v_j, v_{j+1}) with $v_{j+1}-v_j$ equal to the size of the maximum gap is said to be a *maximum gap* of V at *position* v_j.

We first show that two events (E1) and (E2) occur with sufficiently high probability.

(E1) The maximum gap of V is unique, i.e., it occurs at exactly one position.

To prove this, we bound the probability that, when k values $w_1, \cdots, w_k$ are chosen randomly from $[1, n^4]$, the multiset of gap sizes $\{w_{j+1}-w_j \mid 1 \leqq j \leqq k\}$ contains no repeated sizes, where the values have been sorted and $w_{k+1}=w_1+n^4$ as before (we say that the set has *unique gaps*). Let u_k be the probability that a random set of k values does not have unique gaps. It is obvious that $u_1=0$. To bound u_k for $k>1$, first randomly choose a set R of $k-1$ values. With probability at most u_{k-1}, R does not have unique gaps. In the case that R has unique gaps, in order to produce nonunique gaps the kth value must either split a gap of R exactly in half (there are at most $k-1$ ways to do this) or split a gap to create a new gap with the same size as one of the gaps of R (there are at most $2(1+2+\cdots+(k-2))$ ways to do this). Therefore,

$$u_k \leqq u_{k-1}+k^2 n^{-4},$$

so $u_n \leqq nn^2 n^{-4}=n^{-1}$.

(E2) $g=2n^3 \ln n+2$ is an upper bound on the size of the maximum gap of V.

Let $s=\lceil n^3 \ln n \rceil$, let m be the largest multiple of s such that $m \leqq n^4$, and partition the set $\{1, 2, \cdots, m\}$ into *cells*, where each cell consists of s consecutive integers. If $m<n^4$, create another cell containing $m+1, m+2, \cdots, n^4, 1, 2, \cdots, s+m-n^4$. We say that a cell X is *empty* if X contains no value in V. Fix an arbitrary cell X. The probability that a randomly chosen value falls in X is $s/n^4 \geqq (\ln n)/n$. Therefore,

$$\begin{aligned}\Pr[X \text{ is empty}] &\leqq \left(1-\frac{\ln n}{n}\right)^{n-t} \\ &\leqq e^{-((\ln n)/n)(n-t)} \\ &= e^{-\ln n}\, e^{(t \ln n)/n} \\ &\leqq n^{-1}\, e^{1/6}.\end{aligned}$$

Since the number of cells is $\lceil n^4/s \rceil \leqq n/\ln n+1$,

$$\Pr[\text{some cell is empty}] \leqq \frac{e^{1/6}}{n}\left(\frac{n}{\ln n}+1\right)=e^{1/6}\left(\frac{1}{\ln n}+\frac{1}{n}\right) \leqq \frac{1}{2}.$$

The last inequality holds since we have assumed $n \geqq 17$. Since the desired upper bound g on the size of the maximum gap is at least twice the size s of each cell, if the size of the maximum gap exceeds g then some cell must be empty. Therefore, g is an upper bound on the size of the maximum gap with probability at least $\frac{1}{2}$.

For the remainder of the proof, we assume that (E1) and (E2) occur and all probabilities are conditioned on these events occurring. Call the unique maximum gap of V the *correct gap.*

Recall that the *position* of the correct gap is the value that is the left endpoint of the correct gap. Say that a choice w for the position of the correct gap is *bad* for the correct processor p_j if one of the t values a_{ij}, $1 \leqq i \leqq t$, satisfies $w \leqq a_{ij} < w+g$ or

$a_{ij} < w+g-n^4$ (in the case that $w+g > n^4$); otherwise, w is *good* for p_j. Say that a position w is *persuasive* if w is good for some set C of at least $\lfloor n/2 \rfloor + t + 1$ correct processors. The key fact is that if w is persuasive, then all processors in C choose the same leader (the smallest numbered processor that sent value w) and this leader is correct. Moreover, this holds even after erasing any of the values a_{ij}, since erasing an a_{ij} outside of the correct gap cannot create a new gap of size greater than or equal to the size of the correct gap, and therefore such an erasure will not change the leader chosen by the processors in C.

We now show by a simple counting argument that some positive fraction of the possible positions for the correct gap are persuasive. For each correct p_j, at most gt positions are bad for p_j. Therefore, there are at most $gt(n-t)$ pairs (w, j) such that position w is bad for the correct processor p_j. Assume for contradiction that the fraction $6c$ of the n^4 possible positions are *not* persuasive. Since each nonpersuasive position must be bad for at least $n/2-2t$ correct processors, there are at least $6cn^4(n/2-2t)$ pairs (w, j) such that w is bad for p_j. Comparing these upper and lower bounds on the number N of such pairs,

$$6cn^4(n/2-2t) \leqq N \leqq gt(n-t) \leqq (2cn^4+2t)(n-t).$$

Since $n \geqq 17$ implies $t < n/12$, the leftmost expression is greater than $2cn^5$. Assuming $t \geqq 1$, a calculation shows that the rightmost expression is less than $2cn^5$. This gives a contradiction. In other words, at least the positive fraction $1-6c$ of the positions are persuasive.

Since we assume (E1) that there is a unique position for the correct gap, it is easy to see by symmetry that the position of the correct gap is uniformly distributed in $[1, n^4]$. (Recall that we have bent the interval $[0, n^4]$ into a circle.) Therefore a persuasive position for the correct gap will be chosen with probability at least $1-6c$. □

Remarks. (1) The proof shows that any probability $\rho < \frac{1}{2}$ is possible by making c sufficiently small.

(2) Except for some improvement in the constant c, the resiliency of Algorithm COIN cannot be improved if the probability ρ should be bounded above zero. A good strategy for the adversary is to spread the t values of the t faulty processors evenly across the interval $[1, n^4]$, creating $t-1$ cells each of length $\lceil n^4/t \rceil$ (and one cell of possibly smaller length). In order to prevent a faulty processor from being chosen as the leader by all correct processors, a value from a correct processor must fall into every one of these $t-1$ cells. If $t \geqq cn/\ln n$ for some constant $c > 1$, then the probability of this event goes to zero as n increases (see, for example, [19, § IV.2]).

3.2. Rushing without eavesdropping. We now modify the previous algorithm to withstand any adversary in the rushing without eavesdropping model. Note that in this stronger model the old algorithm is not even 1-resilient. The adversary makes p_1 faulty during the first round, collects the messages sent by correct processors to p_1, finds the leftmost maximum length gap, and instructs p_1 to send a value equal to the left endpoint of this maximum gap. This adversary controls the coin with probability 1.

The key tool that we develop to overcome this difficulty is a broadcast primitive that we call *simultaneous broadcast with erasing*. In this primitive we imagine that all processors simultaneously choose a message; let m_i be the message chosen by p_i. Then all messages $m_1, \cdots, m_n$ are shown to the faulty processors. For each correct p_i and each faulty p_j, the faulty p_j can choose either to send m_j to p_i or to send no message to p_i; this choice can be different for each pair of correct p_i and faulty p_j. Each correct p_i sends m_i to all processors.

For this primitive, we use the secret-sharing technique of Shamir [25]. The aim of this technique is to split a secret into n shares so that any t shares do not give any information about the secret, but any $t+1$ shares suffice to compute the secret efficiently. Suppose that a secret, encoded as a positive integer v, is to be split into n shares. Let q be a prime number larger than v and n, choose random $r_1, r_2, \cdots, r_t \in Z_q$ (the field of integers modulo q), and let $Q(x)$ be the polynomial over Z_q of degree t,

$$Q(x) = r_t x^t + \cdots + r_1 x + v.$$

Note that $Q(0) = v$. The shares are then $Q(1), \cdots, Q(n)$. Given $t+1$ shares of the form $(i, Q(i))$, the unique degree-t polynomial Q that can be interpolated through these points can be found in polynomial time, and then the secret, $Q(0)$, can be computed. In addition, it is easy to see that given fewer shares there are an equal number of degree-t polynomials that interpolate to yield constant term v for every $v \in Z_q$.

The following procedure requires that $n > 4t$.

ALGORITHM BROADCAST

Algorithm for processor p_i:

1. During round 1, choose the message m_i, split it into n shares, $s_i(1), \cdots, s_i(n)$, and send the share $s_i(j)$ to processor p_j.
2. During round 2, broadcast the vector of shares $w_i = \langle s_i(1), \cdots, s_i(n) \rangle$ and all shares received during round 1.
3. For each j, compute the message m_j:
   - 3.1. For every k, two views of $s_j(k)$ should be received in round 2, one in the vector w_j and one forwarded from p_k. Let S_{ji} be the set of processors p_k for which the two views are identical.
   - 3.2. If $|S_{ji}| < n - t$ or if no degree-t polynomial can be interpolated through w_j, then no message was received from p_j. Otherwise, if Q is the interpolating polynomial, then $m_j = Q(0)$.

LEMMA 3.2. *Let n and t be such that $n > 4t$ and assume the model of rushing without eavesdropping. Algorithm* BROADCAST *is a t-resilient simultaneous broadcast with erasing primitive.*

Proof. First we show that a correct processor p_j sends m_j to all (correct) processors. For any processor p_i computing the message from p_j, the set S_{ji} contains all the correct processors. Since there are at least $n - t$ of them, and since w_j is computed correctly, processor p_i will receive message m_j.

Next we show that the messages received in round 1 and the choice of faulty processors for round 2 determine, for each faulty processor p_j, a unique message m_j such that any correct processor either receives m_j from p_j in Step 3.2 or receives no message from p_j. Consider two computations in Step 3 that yield nonnull messages and let S and S' denote the sets found in Step 3.1 using the vectors w_j and w'_j. We give a counting argument which shows that there must be at least $t+1$ coordinates in which w_j and w'_j are equal. Since $|S| \geqq n - t$ and $|S'| \geqq n - t$, there are at least $n - 2t$ processors in $S \cap S'$. Furthermore, if a processor is in $S \cap S'$, then either the corresponding values in w_j and w'_j are equal, or the processor is faulty. Since there are at most t faulty processors, there are at least $(n - 2t) - t = n - 3t > t$ coordinates in which w_j and w'_j are equal. However, $t+1$ coordinates define a unique degree-t polynomial, so that the results of interpolating w_j and w'_j must be equal.

To complete the proof, it is only necessary to note that by the properties of Shamir's technique, the messages sent by correct processors in round 1 give the t faulty

processors no information about the message values sent by correct processors. Therefore, values "locked in" during round 1 by the faulty processors are selected independently of the correct processors' messages. □

We modify Algorithm COIN given above by replacing the ordinary broadcast in Step 2 with a broadcast using the new primitive. The proof that this new protocol works within the rushing without eavesdroppping model is identical to the proof of Theorem 3.1.

THEOREM 3.3. *Let* $c < \frac{1}{6}$ *and assume the model of rushing without eavesdropping.*

(1) *There is a* 2*-round* $(cn/\ln n)$*-resilient persuasive coin with nonzero constant probability.*

(2) *There is a* $(cn/\ln n)$*-resilient probabilistic Byzantine agreement algorithm such that the expected number of rounds to reach agreement is constant.*

(*The constants in* (1) *and* (2) *are independent of* n.)

4. Cryptographic algorithms with constant expected running time. In this section we strengthen the protocols of § 3 to tolerate the blocking adversary. Recall that in choosing at each round r which processors fail in round r the adversary can make this choice, and the choice of round r messages to be sent by faulty processors, based on the messages sent by all processors in all rounds up to and including r. Thus, the blocking adversary can eavesdrop on the conversations between correct processors. The basic coin protocol cannot handle eavesdropping because it requires that the values chosen by faulty processors be independent of the values chosen by correct processors. Therefore, cryptography seems to be needed. Because we will be using cryptography, we will assume the adversary is restricted to functions computable by a probabilistic polynomial time bounded Turing machine, where the polynomial is in n, the total number of processors, and k, a security parameter.

The general idea of our solution is to modify the basic coin flipping protocol, encrypting all messages using the probabilistic public key cryptosystems (PPKC's) of Goldwasser and Micali [22]. However, some additional care must be taken. In constructing a modification of the algorithm COIN to work in the presence of a blocking adversary, we might expect that the following simple algorithm would sufficiently protect the values to be broadcast: each p_i picks a single encryption/decryption pair (E_i, D_i), broadcasts E_i and $E_i(v_i, b_i)$ during the first round, and broadcasts D_i during the second round. Surprisingly, this protocol is not even $2\sqrt{n}$-resilient to a blocking adversary.

In the first round, the adversary corrupts $\sqrt{n}$ processors and chooses values v_i that are distributed evenly, every $n^{7/2}$. In the second round, the adversary collects all of the decryption keys and computes all of the (correct) message values. The corrupted values define $\sqrt{n}$ slots, and there is some slot that has no more than $\sqrt{n}$ values of correct processors. By blocking the decryption keys used for the values in the leftmost such slot, the adversary can be guaranteed to control the coin. As this example makes clear, the adversary gains substantial power by choosing the faulty processors for the current round after receiving all of the messages for that round.

Algorithm ENCRYPTED COIN uses the Crusader agreement of Dolev [12] as a communication primitive. This two round protocol solves a weak version of the single source Byzantine agreement problem. In Crusader agreement there is a single source s that wishes to broadcast a value $v \neq \perp$ to every processor in the system. Every processor p has a variable v_p that is initially undefined. A protocol solves Crusader agreement if on completion:

1. If s is correct then for all correct processors p, $v_p = v$.

2. For all pairs p, q of correct processors either $v_p = v_q$ or at least one of v_p, v_q has value $\perp$.

The protocol presented below uses secret-sharing (see § 3.2), strengthened by additional certification that the shares were computed properly. We assume the existence of a probabilistic public key cryptosystem Π, as defined by Goldwasser and Micali [22]. Everything in this protocol is done bit-by-bit. By this we mean that each m-bit secret to be shared is shared as m 1-bit secrets, and each m-bit message to be encrypted is encrypted as m 1-bit messages. Thus, our "message space" is just $\{0, 1\}$. Keeping this in mind we obtain the following definition for a public key cryptosystem:

A *probabilistic public key cryptosystem* (PPKC) is a probabilistic polynomial time Turing machine Π that on input 1^k outputs the description of two algorithms E and D such that

(1) for some constant c both E and D halt within k^c steps,
(2) for $b \in \{0, 1\}$, $D(E(b)) = b$,
(3) E is probabilistic, and
(4) the E/D pair produced is drawn uniformly at random from the range of Π on input 1^k.

We call E the *encryption* algorithm and D the *decryption* algorithm. We let $\Pi(k)$ denote the set of encryption/decryption pairs generated by Π on input 1^k. Let $\lambda(k)$ denote the number of random bits used by $E \in \Pi(k)$ to encrypt a single bit.

In the following, let q be a prime larger than n. When using Shamir's secret-sharing technique with resilience t, processors will choose polynomials of degree t with coefficients chosen from Z_q, and all arithmetic is over Z_q. We take q to be part of the protocol. Let N be the least power of 2 satisfying $N \geqq n^4$. In the cryptographic algorithm the values v_i are drawn uniformly at random from $0, 1, \cdots, N-1$. Thus each v_i has exactly $\log N$ bits, and each bit has value 1 with probability exactly $\frac{1}{2}$.

ALGORITHM ENCRYPTED COIN (security parameter k)
Algorithm for processor p_i:

1. During round 1, for each processor p_j, invoke $\Pi(k)$ to obtain a random encryption/decryption pair (E_{ji}, D_{ji}). Run Crusader agreement on $\langle E_{1i}, E_{2i}, \cdots, E_{ni}\rangle$. This requires two rounds of communication. Let B_{1i} denote the set of processors found faulty by p_i during the executions of Crusader agreement initiated by all processors.
2. During this step, the random values are chosen and broken into shares:
   2.1. Randomly choose a value v_i by tossing an unbiased coin $\log N$ times, and choose a b_i by tossing this coin once. Thus (v_i, b_i) can be represented with $m = 1 + \log N$ bits. Let $(v_i, b_i)_h$ denote the hth bit of (v_i, b_i). For each h, $1 \leqq h \leqq m$, break bit $(v_i, b_i)_h$ into n shares $Q_i^h(1), \cdots, Q_i^h(n)$ using Shamir's secret-sharing technique with resilience t. Each share has $\lceil \log q \rceil$ bits. Let $Q_i^{h,x}(j)$ denote the xth bit of the jth share of the hth bit of (v_i, b_i).
   2.2. Let w_{1i} denote the vector of $nm\lceil \log q \rceil$ components, each of the form $E_{ij}(Q_i^{h,x}(j))$, where $1 \leqq j \leqq n$, $1 \leqq h \leqq m$, $1 \leqq x \leqq \lceil \log q \rceil$, and $p_j \notin B_{1i}$. If $p_j \in B_{1i}$ then the special symbol $\perp$ is used for these components. Let $R_{ij}^{h,x}$ denote the random string used to generate the probabilistic encryption, $E_{ij}(Q_i^{h,x}(j))$ for $p_j \notin B_{1i}$. If $p_j \in B_{1i}$ then choose a random string $R_{ij}^{h,x}$ even though no encryption is performed on the share $Q_i^{h,x}(j)$. Recall that each $R_{ij}^{h,x}$ has length $\lambda(k)$.
   2.3. Let w_{2i} denote the vector consisting of each plaintext bit encrypted in w_{1i} together with the random string used in the encryption. That is, $w_{2i} =$

$\langle R_{i1}^{1,1}, Q_i^{1,1}(1), \cdots, R_{in}^{m,\lceil \log q \rceil}, Q_i^{m,\lceil \log q \rceil}(n)\rangle$. w_{2i} contains $2nm\lceil \log q \rceil$ components and $m' = nm\lceil \log q \rceil(\lambda(k)+1)$ bits. Split each bit h of this vector into n shares, $S_i^h(1), \cdots, S_i^h(n)$, $1 \leq h \leq m'$. Each share contains $\lceil \log q \rceil$ bits. Let $S_i^{h,x}(j)$ denote the xth bit of $S_i^h(j)$.

2.4. Let w_{3i} denote the vector of $nm'\lceil \log q \rceil$ components $E_{ij}(S_i^{h,x}(j))$, where $1 \leq j \leq n$, $1 \leq h \leq m'$, $1 \leq x \leq \lceil \log q \rceil$, and $p_j \notin B_{1i}$. If $p_j \in B_{1i}$ then the special symbol $\perp$ is used for these components. Broadcast the vectors w_{1i} and w_{3i}.

3. During Step 3, broadcast the decryption keys $D_{1i}, \cdots, D_{ni}$ and then compute the value (v_j, b_j) received from each processor p_j as follows:

3.1. Decrypt those components $E_{jy}(S_j^{h,x}(y))$ of w_{3j} for which D_{jy} was received to obtain the xth bit of the yth share $S_j^h(y)$ of the hth bit of w_{2j}. For each h, $1 \leq h \leq m'$, interpolate the decrypted shares $S_j^h(y)$ to find the bits of w_{2j}. Verify that there are at most t processors p_f such that either p_j replaced $E_{jf}(S_j^{h,x}(f))$ with $\perp$ or $p_f \in B_{1i}$.

3.2. Decrypt those components $E_{jy}(Q_j^{h,x}(y))$ of w_{1j} for which D_{jy} was received to obtain the yth share $Q_j^h(y)$ of $(v_j, b_j)_h$. For each h, $1 \leq h \leq m$, interpolate the decrypted shares $Q_j^h(y)$ to find the bits of (v_j, b_j). Verify that there are at most t processors p_f such that either p_j replaced $E_{jf}(Q_j^{h,x}(f))$ with $\perp$ or $p_f \in B_{1i}$.

3.3. From w_{2j}, verify that all non-$\perp$ components $E_{jy}(Q_j^{h,x}(y))$ of w_{1j} for processors $p_y \notin B_{1i}$ were computed correctly by p_j, where $1 \leq h \leq m$. Verify that for each h a degree-t polynomial can be interpolated through the shares $Q_j^h(s)$, $s = 1, \cdots, n$ that are given in w_{2j}.

3.4. If any interpolation or verification fails, then no (nonempty) message was received from p_j.

4. Compute the value of the coin using the nonempty messages as described in the algorithm COIN.

Let $t = cn/\log n$ be an upper bound on the number of faulty processors in any execution of Algorithm ENCRYPTED COIN.

Fix an execution of Algorithm ENCRYPTED COIN. Let B_1 denote the set of processors made faulty during Step 1 of the execution, and let G_1 be the complement of B_1. Similarly, let $B_2 \supseteq B_1$ denote the set of processors made faulty during Steps 1 and 2, and let $G_2 \subseteq G_1$ denote the complement of B_2.

Let us consider informally how the adversary might force a particular outcome of the coin, say 0. If the bit associated with the left endpoint of the maximum gap among the values chosen by G_1 is 0, then the adversary need only stay out of the maximum gap. On the other hand, if this bit is 1, then the adversary must gain control of the coin. To do this, it could selectively kill members of G_1, inducing a set G_2 of correct processors such that the bit associated with the left endpoint of the maximum gap among the values chosen by G_2 is 0. Alternatively, it could prevent the coin from being persuasive or make a faulty processor leader, both of which require it to land a value inside the correct gap. Our intuition is that if the values (v_i, b_i) of processors in G_1 are encrypted, then the adversary has no knowledge of these values. In this case the adversary must choose B_2 (and hence G_2), and choose values for processors in B_2, without any information about the values of processors in G_1. But this is precisely the situation in Algorithm COIN.

Let predicates (E1) and (E2) be as in the proof of Theorem 3.1. Recall that (E1) says the correct gap, defined by the values of processors in G_2, is unique, and (E2) says this gap is not too large.

Let P_0 be true if and only if, in addition to (E1) and (E2), the bit of the leader defined by the correct gap is 0, and the position of the correct gap is good for at least $n/2+2cn/\log n+1$ processors in G_2 (i.e., no processor in B_2 lands a value in this gap in the view of these $n/2+2cn/\log n+1$ processors).

Let P_1 be defined analogously for the case in which the bit of the leader defined by the correct gap is 1. In the following, our claims about P_0 hold for P_1 as well.

We will define a chain of simulations of the first two steps of the algorithm, such that at one end of the chain the adversary has no information about the values of processors in G_1, and therefore the predicate P_0 holds with the same probability as it holds in protocol COIN when run in the presence of $2cn/\log n$ faults. At the other end of the chain, the predicate P_0 is no less likely to hold than in an execution of Algorithm ENCRYPTED COIN in the presence of $2cn/\log n$ faults. Moreover, in the Claim below, we argue that the probabilities that P_0 holds at the two ends of the chain are extremely close (defined more precisely in the statement of the Claim).

By Theorem 3.1, in each execution of Algorithm COIN, both P_0 and P_1 hold with probability at least ρ, where ρ is a fixed constant. Combining this with the Claim we will see that there exists a constant ρ' such that both P_0 and P_1 hold with probability at least ρ' after the first two steps of Algorithm ENCRYPTED COIN. Suppose P_0 holds before the adversary has chosen which processors to make faulty in Step 3. Then at least $n/2+2cn/\log n+1$ processors in G_2 see an outcome of 0 for the coin. Thus, if G_3 is the set of processors remaining correct through Step 3, then at least $n/2+cn/\log n+1$ correct processors see a 0 at the end of the execution of the coin flipping protocol, and the flip is indeed persuasive.

Before we begin our proof of correctness of Algorithm ENCRYPTED COIN we review some of the properties of probabilistic public key cryptosystems. These properties and definitions have been adapted from the more general setting of Goldwasser and Micali [22]. We restrict ourselves to bit-by-bit encryption. Thus the only messages ever encrypted are 0 and 1.

Let Π be a PPKC. Let $\mathcal{T}$ be a probabilistic Turing machine that, when given as input (k, E, γ), where E is a description of an encryption algorithm $E \in \Pi(k)$, and $\gamma \in E(m)$ for some $m \in \{0, 1\}$, produces a single binary output. Such a machine is called a *line tapper.* Let $E \in \Pi(k)$ be fixed. Given a plaintext message m, there is an a priori probability, taken over all possible encryptions $E(m)$ and all coin tosses of $\mathcal{T}$, that $\mathcal{T}(k, E, E(m))=1$. Let $p(E, m)$ denote this probability. Clearly, $p(E, 0)$ and $p(E, 1)$ may differ. In particular, if $|p(E, 0)-p(E, 1)|>1/P(k)$, then we say that $\mathcal{T}$ *P-distinguishes* 0 *and* 1 *with respect to E.*

Π is *polynomially secure* if for all polynomials R and for every infinite set I of positive integers, there does not exist a probabilistic Turing machine $\mathcal{T}$, having running time bounded by a polynomial in k, which, for all $k \in I$, R-distinguishes 0 and 1 with respect to any polynomial fraction of the encryption keys $E \in \Pi(k)$. Goldwasser and Micali [22] show that if the quadratic residuosity problem cannot be solved by a probabilistic Turing machine in polynomial time, then a polynomially secure PPKC exists.

Let n the number of players, and $k \geqq n$ the security parameter, be fixed. We define a sequence, or "chain" of simulators. Each simulator has two players, the *System* and the *Adversary*, denoted S and A, respectively. Intuitively, the system player corresponds to a system of n processors executing Algorithm ENCRYPTED COIN with security parameter k. The adversary is the blocking adversary. In each simulation, the system player simulates the internal state changes of all uncorrupted processors. The first simulator actually simulates executions of the algorithm, with the system player sending

to the adversary all the ciphertext messages of uncorrupted processors, the adversary choosing processors to corrupt and choosing messages of those corrupted. In the final simulator the system player sends encryptions of random noise to the adversary. The adversary again chooses processors to corrupt and chooses messages for those corrupted. Recall that all our encryptions are of single bits. The intuition behind the chain is that each successive simulator replaces one additional encryption of a real bit, determined by the internal state of some simulated correct processor, by the encryption of a bit chosen at random. We describe the simulators more precisely below.

Let us first compute the length of the chain and devise a method of naming each simulator. As above, B_1 will denote the set of processors corrupted by the adversary in Step 1 of the algorithm. Recall that Step 1 is composed of two rounds of message exchange. To determine the length of the chain we must compute the total number of encryptions of single bits to which the adversary has access in an execution in which B_1 is empty. Consider a correct processor p_i. During Step 2, p_i broadcasts vectors w_{1i} and w_{3i}, containing $n\lceil \log q \rceil m$ and $n\lceil \log q \rceil m'$ distinct encryptions, respectively. Thus, if B_1 is empty the adversary has access to $n^2\lceil \log q \rceil (m+m')$ separate encryptions of bits in Step 2. The length of the chain is therefore $1+n^2\lceil \log q \rceil (m+m')$. Recall that the intuition behind the chain is that each successive simulator replaces one "real" encryption by the encryption of a random bit. Thus, we can name a simulator by naming the new encryption to be replaced. In the first simulator there is no such bit. We name this machine M_0. In each of the remaining simulators the bit is identified by a 5-tuple $\langle i, j, h, x, B\rangle$, where i and j are a pair of processors, $1 \leqq x \leqq \lceil \log q \rceil$, $B \in \{Q, S\}$ and $h \leqq n\lceil \log q \rceil m$ if $B = Q$, $h \leqq n\lceil \log q \rceil m'$ if $B = S$. The 5-tuple $\langle i, j, h, x, B\rangle$ names the bit $B_i^{h,x}(j)$ (the xth bit of the jth share of B_i^h). Thus, if $B = Q$, then $B_i^{h,x}(j)$ is the xth bit of the jth share of the hth bit of (v_i, b_i), while if $B = S$, then $B_i^{h,x}(j)$ is the xth bit of the jth share of the hth bit of w_{2i}. The simulators are ordered lexicographically, according to their 5-tuple names.

We now describe the simulators precisely. The simulators all handle Step 1 of the simulation of Algorithm ENCRYPTED COIN identically. They differ only in their treatment of Step 2, the step in which ciphertext messages are sent. On input 1^k, S begins by simulating all n processors, first choosing all n^2 encryption/decryptions pairs (E_{ji}, D_{ji}) by repeatedly invoking $\Pi(k)$. Once these choices are made S sends all the E_{ij} to A. A chooses a set B_{1a} of faulty processors to corrupt and chooses the round 1 messages of these processors. These choices are reported to S. S then sends to A the round 2 messages of the uncorrupted processors. These messages may depend on the choices of round 1 messages sent to S by A for members in B_{1a}. Given the round 2 messages for processors not in B_{1a}, A chooses a set B_{1b} of additional processors to be corrupted in round 2, subject to the condition that $|B_{1a} \cup B_{1b}| \leqq t$. A also chooses the round 2 messages of the processors in $B_1 = B_{1a} \cup B_{1b}$. These choices are all reported to S. This completes the description of Step 1 for all simulators in the chain.

We now describe the simulations of Step 2. In the first simulator, M_0, S simulates the behavior of all processors $p_i \notin B_1$. Thus, for each such p_i, S chooses a v_i, b_i pair according to the protocol, breaks each bit $(v_i, b_i)_h$, $1 \leqq h \leqq m$, into shares $Q_i^h(1), \cdots, Q_i^h(n)$, computes w_{1i}, w_{2i}, and w_{3i}, and finally sends w_{1i} and w_{3i} to A. A chooses a set $B_2 - B_1$ of processors to corrupt in Step 2 ($|B_2| \leqq t$) and chooses Step 2 messages for B_2. These choices are sent to S.

In the simulator named by $\langle i, j, h, x, B\rangle$, $1 \leqq i, j \leqq n$, $B \in \{Q, S\}$, $1 \leqq h \leqq m$ if $B = Q$, $1 \leqq h \leqq m'$ if $B = S$, and $1 \leqq x \leqq \lceil \log q \rceil$, S computes the vectors w_{i1}, w_{i2}, and w_{i3} for all $p_i \notin B_1$, as above, but it modifies the messages it sends to A as follows. For each 5-tuple $\langle u, v, g, y, C\rangle$ in our naming scheme such that $\langle u, v, g, y, C\rangle$ either equals

$\langle i, j, h, x, B\rangle$ or precedes $\langle i, j, h, x, B\rangle$ in the lexicographic ordering of these 5-tuples, if neither u nor v is in B_1, then S replaces $E_{uv}(C_u^{g,y}(v))$ with $E_{uv}(r)$, where r is chosen from $\{0, 1\}$ uniformly at random. A different random bit is chosen for each replaced encryption. As in the first simulation, A chooses processors to corrupt in Step 2 and chooses messages for processors in B_2, the set of all processors made faulty during Steps 1 and 2. These choices are sent to S. Note that these choices may contain conflicting messages if the adversary chooses to make a faulty processor send different messages to different correct processors. This completes the descriptions of the simulators when simulating Step 2 of the algorithm.

The simulators do not differ in their method of evaluating the predicate P_0. For each processor $p_i \notin B_2$, S knows the values (v_i, b_i), because S chose these and A did not change them. These values determine the correct gap. For each processor $p_j \in B_2$, and for each processor $p_i \in G_2$, let $w_{1j,i}$ denote the value of the vector w_{1j} sent by the adversary to processor p_i. (Although it was supposed to broadcast a vector w_{1j}, the faulty p_j could instead have sent different vectors to different processors.) S decrypts the shares of $w_{1j,i}$ belonging to processors $p_y \in G_2$ using the decryption keys D_{jy} that it chose in Step 1 while simulating p_y. If these shares are properly chosen and encrypted, then S can interpolate the decrypted shares to obtain what we will call "the value of (v_j, b_j) sent to p_i." If the correct gap is not unique ($\neg$E1), is too large ($\neg$E2), if the leader specified by the correct gap did not flip a 0, or if the position of the correct gap is not good for at least $n/2+2cn/\log n+1$ processors in G_2, then P_0 does not hold. In this case we say the adversary *wins*, and the simulator outputs 1. Otherwise, the adversary *loses*, and the simulator outputs 0.

We remark that if a processor p_i is corrupted in Step 3, then by our definition the adversary only controls the messages sent by p_i in that step, and we can assume the processor receives all its messages and makes local state changes as if it were uncorrupted. Correctness of the algorithm does not depend on this, but it simplifies the discussion.

We now make a few observations about the chain of simulators and the relation of the chain to executions of Algorithm ENCRYPTED COIN.

(1) There is a natural correspondence between executions of M_0 with adversary A, and executions of the algorithm when run against adversary A. Specifically, every execution of the first two steps of the algorithm is simulated by M_0 with exactly the same probability with which the execution of these two steps occurs in actuality, and conversely. The probability space is the set of random choices made by the correct processors (simulated by S), the adversary A, and the PPKC Π. Once M_0 has simulated the first two steps of the algorithm, the rest of the simulation (evaluation of P_0) is fixed. Thus, in talking about an execution of the algorithm, we may refer to "*the* corresponding execution" of M_0, meaning that execution of M_0 agreeing with the first two steps of the actual execution of the algorithm. However, even though the processors behave deterministically in Step 3, the adversary can extend an actual execution of the first two steps of the protocol in many ways. Thus, given an execution of the simulator M_0 there are many corresponding executions of algorithm.

(2) Fix an execution of M_0, and let p_j be a processor in G_2. Let (v_j, b_j) be the value chosen by S for p_j in the simulation of Step 2.1. Then S uses v_j in computing the correct gap when evaluating P_0. Moreover, in any corresponding execution of the algorithm, all $p_i \in G_2$ receive (v_j, b_j) in Step 3, as we now explain. If $p_j \in G_2$, then it correctly prepares and broadcasts its vectors w_{1j} and w_{3j}, and by definition of G_2 these messages are not corrupted by the adversary. Moreover, because p_j correctly prepares these vectors, for each bit $(v_j, b_j)_h$ the component $E_{jy}(Q_j^h(y))$ is $\perp$ only if $y \in B_1$. In Step 3 the adversary can only prevent the dissemination of decryption keys of faulty

processors. For each bit $(v_j, b_j)_h$ the total number of shares $Q_j^h(y)$ unavailable to p_i, either because p_j replaced the share with $\perp$ or because the adversary suppressed D_{jy}, is bounded by $cn/\log n$, the total number of faults. An inspection of Step 3 indicates that the absence of these shares cannot prevent p_i from receiving (v_j, b_j).

(3) Fix an execution of the algorithm, and let p_j be in B_2. Let p_i be in G_2, and let (v_{ji}, b_{ji}) denote the value of (v_j, b_j) computed by p_i in Step 3. If (v_{ji}, b_{ji}) is not empty, then in the corresponding execution of M_0, (v_{ji}, b_{ji}) is the value of (v_j, b_j) sent to p_i. In other words, if a value from p_j gets through in an actual execution, it gets through in the corresponding simulation. This is because if (v_{ji}, b_{ji}) is not empty in the actual execution, then in Step 3.3 p_i was able to verify that all non-$\perp$ components $E_{jy}(Q_j^{h,x}(y))$ of w_{1j} for processors $p_y \notin B_{1i}$ were computed correctly, and that not too many shares of each bit were replaced by $\perp$. Thus, the shares of each bit of (v_{ji}, b_{ji}) can be interpolated by S in the corresponding execution of M_0, so (v_{ji}, b_{ji}) is the value of (v_j, b_j) sent to p_i.

(4) If P_0 holds in an execution of M_0, then it holds in any corresponding execution E of the actual algorithm. By Observation (2), all values of processors in G_2 are the same in corresponding executions, so those clauses of P_0 discussing only the values of processors in G_2 hold in the execution of the algorithm. By Observation (3), if p_i receives a value from $p_j \in B_2$ in E, then it gets through in the corresponding simulation as well. Thus, if the position of the correct gap is good for a processor in G_2 in the simulation, then it will be good for this processor in the execution of the algorithm.

(5) In executions of the last simulator M_L, the set $B_2 - B_1$ and the messages for processors in B_2 are chosen without any information about the Step 2 messages of processors in G_1. This is because in M_L the messages from processors in G_1 to processors in G_1 are replaced by encryptions of random values. Messages from processors in G_1 to processors not in G_1 are not altered in M_L. However, each such message is a share of some bit. So even if the processors not in G_1 combine all the messages they receive from processors in G_1, the messages can be grouped into at most t shares of some bit b_1, at most t shares of some bit b_2, etc. By the property of secret sharing, $z \leqq t$ shares of some bit have the same distribution as z independent random numbers. So in this case, messages in M_L from processors in G_1 are random numbers. Thus, the situation is as in the lock-step model, and P_0 and P_1 hold in executions of M_L with the same probabilities that they hold in protocol COIN. Specifically, each of these holds with probability at least ρ.

In light of Observations (4) and (5), it remains only to prove that P_0 holds in M_0 with probability essentially as large as the probability with which it holds in M_L. We now make this more precise.

Let $L = n^2\lceil \log q \rceil (m + m')$. For each l, $1 \leqq l \leqq L$, let M_l denote the simulator named by the lth 5-tuple in our naming system. As before, M_0 denotes the first simulator in the chain. We allow each simulator a single unary input. On input 1^k the simulator runs its simulation with security parameter k. Each M_l has built into its finite control the finite controls of Π, A, and of S, the system player described above. As described above, each execution of M_l results in a single bit of output. If P_0 does not hold, the adversary wins, and the simulator outputs 1. Otherwise (i.e., if P_0 holds), the adversary loses, and the simulator outputs 0. Each execution of M_l runs in time polynomial in k, because S is simulating only n polynomial (in k) bounded players and A by assumption is polynomial time bounded.

We will need the following notation. For each l, $1 \leqq l \leqq L$, if the lth 5-tuple is $\langle u, w, h, y, B\rangle$, then we let x_l and E_l denote $B_u^{h,y}(w)$ and E_{uw}, respectively. Note that machines M_l and M_{l-1} differ only in their treatment of x_l: in an execution of M_l, with

probability $\frac{1}{2}$, S sends to A an encryption $E_l(x_l)$, while with probability $\frac{1}{2}$, S sends $E_l(1-x_l)$. By contrast, in every execution of M_{l-1}, S sends $E_l(x_l)$. For each k, associated with each M_l is an a priori probability of producing a 1, where the probabilities are taken over all the coin flips of Π, A, and S. Let $\Pr[M_l(k)=1]$ denote this probability.

CLAIM. Assume Π is a polynomially secure PPKC. For all polynomials $P(k)$ and for all probabilistic polynomial time bounded A (polynomial in k), there exists a k_0 such that for all $k \geqq k_0$, if A is used as the adversary player in each M_l then

$$\Pr[M_0(k)=1]-\Pr[M_L(k)=1]<\frac{1}{P(k)}.$$

The proof is by contradiction. We assume the claim is false and show a violation of the polynomial security of Π. Thus, for the sake of contradiction, we let A be a probabilistic polynomial time bounded Turing machine such that for some polynomial P and for all k in some infinite set I,

$$\Pr[M_0(k)=1]-\Pr[M_L(k)=1]\geqq\frac{1}{P(k)}.$$

We will prove the following statements.

1. If $k \in I$ then there exists l, $1 \leqq l \leqq L$, such that

$$\Pr[M_{l-1}(k)=1]-\Pr[M_l(k)=1]\geqq\frac{1}{LP(k)}.$$

2. Let l, $1 \leqq l \leqq L$, k, and ε satisfy

$$\Pr[M_{l-1}(k)=1]-\Pr[M_l(k)=1]\geqq\varepsilon.$$

Then there exists $b \in \{0,1\}$ such that

$$\Pr[M_{l-1}(k)=1 \mid x_l=b]-\Pr[M_l(k)=1 \mid x_l=b]\geqq\varepsilon.$$

3. Let l, $1 \leqq l \leqq L$, $b \in \{0,1\}$, k, and ε satisfy

$$\Pr[M_{l-1}(k)=1 \mid x_l=b]-\Pr[M_l(k)=1 \mid x_l=b]\geqq\varepsilon.$$

Let $\mathscr{E}(k,l,b,\varepsilon) \subseteq \Pi(k)$ be the set of encryption keys $E \in \Pi(k)$ satisfying

$$\Pr[M_{l-1}(k)=1 \mid x_l=b \wedge E_l=E]-\Pr[M_l(k)=1 \mid x_l=b \wedge E_l=E]\geqq\varepsilon/2.$$

Then $|\mathscr{E}(k,l,b,\varepsilon)|/|\Pi(k)|>\varepsilon/2$.

Later, we will need the following notation:

$$\mathscr{E}(k,\varepsilon)=\bigcup_{l,b}\mathscr{E}(k,l,b,\varepsilon),$$

where $1 \leqq l \leqq L$, and $b \in \{0,1\}$.

4. Let M_l, $1 \leqq l \leqq L$, $b \in \{0,1\}$, k, and $E \in \Pi(k)$ be fixed. Let

$$q_0=\Pr[M_{l-1}(k)=1 \mid x_l=b \wedge E_l=E],$$

$$q_1=\Pr[M_l(k)=1 \mid x_l=b \wedge E_l=E].$$

Let

$$r_b=\Pr[M_l(k)=1 \mid x_l=b \wedge E_l=E \wedge S \text{ sends } E_l(x_l) \text{ to } A],$$

$$r_{1-b}=\Pr[M_l(k)=1 \mid x_l=b \wedge E_l=E \wedge S \text{ sends } E_l(1-x_l) \text{ to } A].$$

Then

$$q_0 - q_1 = \tfrac{1}{2}(r_b - r_{1-b}).$$

After proving these statements we will describe our line tapper $\mathcal{T}$. We will then prove the following statement.

5. There exists some polynomial R such that for all $E \in \mathcal{E}(k, 1/LP(k))$, all $k \in I$, and all $\gamma \in E(0) \cup E(1)$,

$$\Pr[\mathcal{T}(k, E, \gamma) = 1 \mid \gamma \in E(1)] - \Pr[\mathcal{T}(k, E, \gamma) = 1 \mid \gamma \in E(0)] \geqq \frac{1}{R(k)}.$$

Let k be in I. By Statements 1-3 the set of keys $\mathcal{E}(k, 1/LP(k))$ has cardinality at least $|\Pi(k)|/2LP(k)$. By Statement 5 there is a polynomial R such that $\mathcal{T}$ R-distinguishes $E(0)$ from $E(1)$ for each encryption key $E \in \mathcal{E}(k, 1/LP(k))$. If in addition $\mathcal{T}$ runs in time polynomial in k, then we have obtained the desired contradiction to the assumed polynomial security of Π. We now proceed according to this outline. The reader may wish to skip the detailed proofs of Statements 1-4 in a first reading, and proceed directly to the description of the line tapper, without any loss in understanding of the structure of the proof of the theorem.

STATEMENT 1. *If $k \in I$ then there exists l, $1 \leqq l \leqq L$, such that*

$$\Pr[M_{l-1}(k) = 1] - \Pr[M_l(k) = 1] \geqq \frac{1}{LP(k)}.$$

Proof. If $k \in I$ then by definition of I

$$\Pr[M_0(k) = 1] - \Pr[M_L(k) = 1] \geqq \frac{1}{P(k)},$$

so there exists an l, $1 \leqq l \leqq L$, such that

$$\Pr[M_{l-1}(k) = 1] - \Pr[M_l(k) = 1] \geqq 1/LP(k).$$

Statement 2 says that if there is a drop of size ε between the probability that each of M_{l-1} and M_l outputs 1, then there is a value $b \in \{0, 1\}$ for x_l that gives rise to a drop of size ε.

STATEMENT 2. *Let l, $1 \leqq l \leqq L$, k, and ε satisfy*

$$\Pr[M_{l-1}(k) = 1] - \Pr[M_l(k) = 1] \geqq \varepsilon.$$

Then there exists $b \in \{0, 1\}$ such that

$$\Pr[M_{l-1}(k) = 1 \mid x_l = b] - \Pr[M_l(k) = 1 \mid x_l = b] \geqq \varepsilon.$$

Proof. We write

$$\Pr[M_{l-1}(k) = 1]$$
$$= \Pr[x_l = 0] \Pr[M_{l-1}(k) = 1 \mid x_l = 0] + \Pr[x_l = 1] \Pr[M_{l-1}(k) = 1 \mid x_l = 1].$$

Similarly,

$$\Pr[M_l(k) = 1]$$
$$= \Pr[x_l = 0] \Pr[M_l(k) = 1 \mid x_l = 0] + \Pr[x_l = 1] \Pr[M_l(k) = 1 \mid x_l = 1].$$

Substituting these expressions into the premise of Statement 2 yields

$$\Pr[x_l = 0] \Pr[M_{l-1}(k) = 1 \mid x_l = 0] + \Pr[x_l = 1] \Pr[M_{l-1}(k) = 1 \mid x_l = 1]$$
$$- (\Pr[x_l = 0] \Pr[M_l(k) = 1 \mid x_l = 0] + \Pr[x_l = 1] \Pr[M_l(k) = 1 \mid x_l = 1]) \geqq \varepsilon.$$

Rearranging terms yields

$$\Pr[x_l = 0](\Pr[M_{l-1}(k) = 1 \mid x_l = 0] - \Pr[M_l(k) = 1 \mid x_l = 0])$$
$$+ \Pr[x_l = 1](\Pr[M_{l-1}(k) = 1 \mid x_l = 1] - \Pr[M_l(k) = 1 \mid x_l = 1]) \geqq \varepsilon.$$

We claim one of these two differences is at least ε. For brevity, let us write this as

$$\Pr[x_l = 0]\alpha + (1 - \Pr[x_l = 0])\beta \geqq \varepsilon.$$

Without loss of generality, let us assume $\alpha \geqq \beta$. Then we have

$$\Pr[x_l = 0]\alpha + (1 - \Pr[x_l = 0])\alpha \geqq \varepsilon,$$

whence $\alpha \geqq \varepsilon$. Similarly, assuming $\beta \geqq \alpha$ yields $\beta \geqq \varepsilon$. Thus at least one of these two differences equals or exceeds ε. This completes the proof of Statement 2.

Statement 3 provides a lower bound on the size of the set of encryption keys for the "critical" encryption function that yield a probability drop.

STATEMENT 3. *Let* l, $1 \leqq l \leqq L$, $b \in \{0, 1\}$, k, *and* ε *satisfy*

$$\Pr[M_{l-1}(k) = 1 \mid x_l = b] - \Pr[M_l(k) = 1 \mid x_l = b] \geqq \varepsilon.$$

Let $\mathscr{E}(k, l, b, \varepsilon) \subseteq \Pi(k)$ *be the set of encryption keys* $E \in \Pi(k)$ *satisfying*

$$\Pr[M_{l-1}(k) = 1 \mid x_l = b \wedge E_l = E] - \Pr[M_l(k) = 1 \mid x_l = b \wedge E_l = E] \geqq \varepsilon/2.$$

Then $|\mathscr{E}(k, l, b, \varepsilon)|/|\Pi(k)| > \varepsilon/2$.

Proof. Let

$$\alpha_0 = \Pr[M_{l-1}(k) = 1 \mid x_l = b],$$
$$\alpha_1 = \Pr[M_l(k) = 1 \mid x_l = b].$$

By assumption $\alpha_0 - \alpha_1 \geqq \varepsilon$. Let $E_1, E_2, \cdots, E_B$, $B = |\Pi(k)|$, be the possible choices of encryption keys in $\Pi(k)$. For $1 \leqq z \leqq B$, let

$$\alpha_{0z} = \Pr[M_{l-1}(k) = 1 \mid x_l = b \wedge E_l = E_z],$$
$$\alpha_{1z} = \Pr[M_l(k) = 1 \mid x_l = b \wedge E_l = E_z].$$

Let q_z denote the probability with which S chooses E_z for E_l. Then for $i \in \{0, 1\}$

$$\alpha_i = \sum_{z=1}^{B} q_z \alpha_{iz}.$$

Let

$$Z = \{z \mid \alpha_{0z} - \alpha_{1z} \geqq \varepsilon/2\}.$$

Let $\kappa = \sum_{z \in Z} q_z$. Note that, because S chooses the encryption key by executing Π on

1^k, and by condition (4) each key is chosen with equal probability, κ is the fraction of keys yielding a probability drop of size at least $\varepsilon/2$. We have

$$\varepsilon \leqq \alpha_0 - \alpha_1 = \sum_{z \in Z} (\alpha_{0z} - \alpha_{1z}) q_z + \sum_{z \notin Z} (\alpha_{0z} - \alpha_{1z}) q_z.$$

Each $\alpha_{0z} - \alpha_{1z}$ is bounded above by 1, so the first term is bounded above by $\sum_{z \in Z} q_z$, which is precisely κ. Moreover, by definition of Z, if $z \notin Z$ then $\alpha_{0z} - \alpha_{1z} < \varepsilon/2$ (otherwise z would be in Z), so the second term is bounded above by $\varepsilon/2$. We therefore have

$$\varepsilon < \kappa + \varepsilon/2,$$

whence $\kappa > \varepsilon/2$. This completes the proof of Statement 3.

STATEMENT 4. *Let* M_l, $1 \leqq l \leqq L$, $b \in \{0, 1\}$, k, *and* $E \in \Pi(k)$ *be fixed. Let*

$$q_0 = \Pr[M_{l-1}(k) = 1 \,|\, x_l = b \wedge E_l = E],$$

$$q_1 = \Pr[M_l(k) = 1 \,|\, x_l = b \wedge E_l = E].$$

Let

$$r_b = \Pr[M_l(k) = 1 \,|\, x_l = b \wedge E_l = E \wedge S \textit{ sends } E_l(x_l) \textit{ to } A],$$

$$r_{1-b} = \Pr[M_l(k) = 1 \,|\, x_l = b \wedge E_l = E \wedge S \textit{ sends } E_l(1 - x_l) \textit{ to } A].$$

Then

$$q_0 - q_1 = \tfrac{1}{2}(r_b - r_{1-b}).$$

Proof. Half of all executions of $M_l(k)$ are actually legal executions of $M_{l-1}(k)$, because the probability that S sends an encryption of the actual value of x_l to A is exactly $\frac{1}{2}$ in an execution of $M_l(k)$. Moreover, this observation holds even when restricted to executions of the two machines in which $x_l = b$ and $E_l = E$. Thus $r_b = q_0$. Moreover, considering only those executions in which $x_l = b$ and $E_l = E$, we have that the overall probability that M_l produces a 1 is the probability that S sends the right encryption times the probability of a 1 in this first case, plus the probability that S sends the wrong encryption times the probability of a 1 in this second case. Thus,

$$q_1 = \frac{r_b}{2} + \frac{r_{1-b}}{2}.$$

We therefore have

$$q_0 - q_1 = r_b - \left(\frac{r_b}{2} + \frac{r_{1-b}}{2}\right) = \frac{1}{2}(r_b - r_{1-b}),$$

which completes the proof of Statement 4.

We now describe our line tapper. The line tapper will perform sizeable amounts of random sampling. We first describe the two machines without specifying the sample sizes, which are computed from the Chebyshev inequality. The actual computation of these sizes is left to the very end of the proof.

On inputs (k, E, γ), where $\gamma \in E(b)$ for some $b \in \{0, 1\}$, $E \in \Pi(k)$, and $k \in I$, $\mathcal{T}$ computes $4L$ probability estimates by random sampling, two each for M_0 and M_L, and four for each of the $L-1$ other machines.

For each l, $0<l\leqq L$, and for each $b\in\{0,1\}$, let $e_{l,b}$ denote $\mathcal{T}$'s estimate of

$$\Pr[M_l(k)=1\,|\,E_l=E\wedge x_l=b],$$

and let $d_{l,b}$ denote $\mathcal{T}$'s estimate of

$$\Pr[M_{l-1}(k)=1\,|\,E_l=E\wedge x_l=b].$$

$\mathcal{T}$ searches for a probability drop

$$d_{l,b}-e_{l,b}\geqq\frac{1}{2LP(k)}-\frac{2}{kLP(k)}$$

where $0<l\leqq L$. If several such l, b pairs exist, then $\mathcal{T}$ chooses the one giving rise to the largest difference.

$\mathcal{T}$ computes each estimate $e_{l,b}$ by simulating many executions of $M_l(k)$ in which x_l has value b and in which $E_l=E$ (computation of $d_{l,b}$ is handled analogously, simulating M_{l-1}). However, because x_l could be a bit that, say, rarely takes on the value 1 (e.g., x_l could be the high-order bit of a share and most elements of Z_q might have a 0 in this position), it may take too long for $\mathcal{T}$ to generate a large sample of executions in which x_l just happens to take on value 1. Later we explain how to generate random executions of the simulators in which a particular bit has a forced value.

If $\mathcal{T}$ finds no l,b pair such that $d_{l,b}-e_{l,b}$ is sufficiently large, then it outputs 0 and halts. Note that until this point we have not used γ, so up to this point $\mathcal{T}$ is equally likely to succeed in finding a good l,b when $\gamma\in E(0)$ as when $\gamma\in E(1)$. If $\mathcal{T}$ has found a desired pair l,b then it constructs one more execution of $M_l(k)$ in which $E_l=E$, and $x_l=b$, but instead of flipping a coin to determine whether to have S send $E(b)$ or $E(1-b)$ to A, S simply sends γ to A. If A wins, then $\mathcal{T}$ outputs the value b (essentially "guessing" that $\gamma\in E(b)$), while if A does not win in the simulated execution then $\mathcal{T}$ outputs $1-b$.

$\mathcal{T}$ simulates an execution of $M_l(k)$ in which x_l has value b as follows. Let $\langle u, w, h, y, B\rangle$ be the lth 5-tuple. Recall that if $B=Q$, then $x_l=B_u^{h,y}(w)$ denotes the yth bit of the wth share of the hth bit of (v_u, b_u), whereas if $B=S$, then $B_u^{h,y}(w)$ denotes the yth bit of the wth share of the hth bit of w_{2u}. Thus, the forced bit is always a single bit of a share of some special bit. Let α be the special bit to be shared. Let $b\in\{0,1\}$ be fixed. We show how to construct a polynomial π of degree t such that $\pi(0)=\alpha$ and the yth bit of $\pi(w)$ is b. Recall that in the secret sharing all arithmetic is over the finite field Z_q, and all shares are represented by $\lceil\log_2 q\rceil$ bits. Thus a given bit of a share can indeed take on either binary value. Without loss of generality, let us assume $w>t-1$. Let $a_0=\alpha$ and let a_w be any value in Z_q whose yth bit is b. Randomly choose $a_1,\cdots,a_{t-1}\in Z_q$. We write

$$\pi(x)=\sum_{i=0,\cdots,t-1,w}a_i\frac{(x-0)(x-1)\cdots(x-i)'\cdots(x-t+1)(x-w)}{(i-0)(i-1)\cdots(i-i)'\cdots(i-t+1)(i-w)},$$

where $(\ )'$ means this factor *does not* occur. $\pi(x)$ is the desired polynomial. The remaining shares of b can be computed according to this formula.

STATEMENT 5. *There exists some polynomial R such that for all $E\in\mathcal{E}(k,1/LP(k))$, all $k\in I$, and all $\gamma\in E(0)\cup E(1)$,*

$$\Pr[\mathcal{T}(k,E,\gamma)=1\,|\,\gamma\in E(1)]-\Pr[\mathcal{T}(k,E,\gamma)=1\,|\,\gamma\in E(0)]\geqq\frac{1}{R(k)}.$$

Proof. If $k \in I$ and $E \in \mathscr{E}(k, 1/LP(k))$, then by Statements 1-3 and the definition of $\mathscr{E}(k, 1/LP(k))$, there exist l,b such that

$$\Pr[M_{l-1}(k) = 1 \mid x_l = b \wedge E_l = E] - \Pr[M_l(k) = 1 \mid x_l = b \wedge E_l = E] \geqq \frac{1}{2LP(k)}.$$

Let c denote the probability that at least one of the $4L$ estimates made by $\mathscr{T}$ differs from the actual probability being estimated by more than $1/kLP(k)$. The number of simulations $\mathscr{T}$ uses to make each estimate will be chosen sufficiently large that c is small. We therefore consider the case in which $\mathscr{T}$ makes no such error. In this case, $\mathscr{T}$ finds a pair l, b such that

$$\Pr[M_{l-1}(k) = 1 \mid x_l = b \wedge E_l = E] - \Pr[M_l(k) = 1 \mid x_l = b \wedge E_l = E] \geqq \varepsilon,$$

where $\varepsilon = 1/2LP(k) - 2/kLP(k)$. We examine the two cases $b = 1$ and $b = 0$ separately.

If $b = 1$, then $\Pr[\mathscr{T}(k, E, \gamma) = 1 \mid \gamma \in E(1)]$ is precisely the probability that the adversary A wins in M_l when S sends to A an encryption of the correct value for x_l. Analogously, $\Pr[\mathscr{T}(k, E, \gamma) = 1 \mid \gamma \in E(0)]$ is the probability that the adversary A wins in M_l when S sends to A an encryption of the wrong value for x_l. By Statement 4 the difference between these two probabilities is at least 2ε.

If $b = 0$, then $\Pr[\mathscr{T}(k, E, \gamma) = 1 \mid \gamma \in E(1)]$ is the probability that the adversary A loses in M_l when S sends to A an encryption of $1 - x_l$, while $\Pr[\mathscr{T}(k, E, \gamma) = 1 \mid \gamma \in E(0)]$ is the probability that A loses in M_l when S sends to A an encryption of x_l. Of course, $\Pr[A \text{ loses}]$ is just $1 - \Pr[A \text{ wins}]$. Combining this with Statement 4 again yields a difference of 2ε.

Putting all this together we have that for $E \in \mathscr{E}(k, 1/LP(k))$ and for all $\gamma \in E(0) \cup E(1)$,

$$\Pr[\mathscr{T}(k, E, \gamma) = 1 \mid \gamma \in E(1)] - \Pr[\mathscr{T}(k, E, \gamma) = 1 \mid \gamma \in E(0)]$$

$$\geqq (1 - c)\left(\frac{1}{LP(k)} - \frac{4}{kLP(k)}\right).$$

Choosing

$$\frac{1}{R(k)} = \frac{1}{2}\left(\frac{1}{LP(k)} - \frac{4}{kLP(k)}\right)$$

proves Statement 5, provided $c \leqq \frac{1}{2}$. Note that $R(k)$ is polynomial in k. It remains only to choose the sample sizes such that c is sufficiently small.

For each estimate obtained by $\mathscr{T}$, let p denote the actual probability that $\mathscr{T}$ is trying to measure. The estimated probability is given by X_m/m, where m is the number of trials (simulations of a particular machine with the value of a specific bit and of a particular encryption function forced) and X_m is the number of these trials in which the adversary wins. We wish to choose m such that

$$\Pr\left[\left|\frac{X_m}{m} - p\right| \geqq \frac{1}{kLP(k)}\right] < \frac{1}{8L}.$$

In this case, the probability that even one of the $4L$ estimates is off by more than $1/(kLP(k))$ is less than $\frac{1}{2}$, as required in the proof of Statement 5. Plugging this into

the Chebyshev inequality

$$\Pr\left[\left|\frac{X_m}{m}-p\right| \geqq \left(\frac{r}{m}\right)^{1/2}\right] < \frac{1}{r}$$

yields $m = 8L(kLP(k))^2$, which is clearly polynomial in k.

Thus $\mathcal{T}$ is a probabilistic polynomial time algorithm such that for a polynomial fraction of the keys $E \in \Pi(k)$ and for all $\gamma \in E(0) \cup E(1)$,

$$\Pr[\mathcal{T}(k, E, \gamma) = 1 \mid \gamma \in E(1)] - \Pr[\mathcal{T}(k, E, \gamma) = 1 \mid \gamma \in E(0)] \geqq \frac{1}{2LP(k)} - \frac{2}{kLP(k)},$$

violating the assumed polynomial security of Π. The Claim is proved.

We note that if we redefine the simulators so that output 1 is produced if and only if P_1 is satisfied, and we redefine *win* so that the adversary wins if and only if P_1 is not satisfied, then the Claim is still true. Thus, under either definition, the adversary is only subpolynomially (in k) more likely to win in M_0 as in M_L, so P_0 and P_1 each hold in M_0 with probability strictly greater than $\rho - 1/P(k)$, for any polynomial P. Combined with Observation (4) above, this says that P_0 and P_1 both hold with probability at least $\rho/2$ in any execution of Algorithm ENCRYPTED COIN run with sufficiently large security parameter k. We therefore have the following theorem.

THEOREM 4.1. *Assume that a polynomially secure* PPKC *exists. Given n, let c be such that Algorithm* COIN *is* $2cn/\log n$*-resilient with probability* ρ*. Then for every polynomial-time blocking adversary A there is a* k_0 *such that, for all* $k \geqq k_0$*, Algorithm* ENCRYPTED COIN (k) *is a* $cn/\log n$*-resilient persuasive coin with probability* $\rho/2$*, resilient to A.*

COROLLARY 4.2. *Assume that a polynomially secure* PPKC *exists. There exist constants* $c, r > 0$ *such that, for every n, there exists a family* $B(k)$ *of n-processor algorithms such that, for every polynomial-time blocking adversary A, there is a* k_0 *such that for all* $k \geqq k_0$*,* $B(k)$ *is a* $(cn/\log n)$*-resilient Byzantine agreement algorithm, resilient to A, where the expected number of rounds to reach agreement is at most r.*

Remark. In these results, the lower bound k_0 on the security parameter depends on the particular adversary A. By modeling adversaries and line tappers as (nonuniform) polynomial-size probabilistic circuits and by changing the definition of a polynomially secure PPKC accordingly, the results can be strengthened so that k_0 depends only on the polynomial that bounds the size of the adversary, i.e., the number of gates in the circuit that computes the behavior of the adversary. After these changes, the key Claim can be modified as follows: For all polynomials $P(k)$ and $Q(k, n)$, there is a k_0 such that for all n, all $k \geqq \max(k_0, n)$, and all A of size at most $Q(k, n)$, if A is used as the adversary in each M_l, then $\Pr[M_0(k) = 1] - \Pr[M_L(k) = 1] < 1/P(k)$. The proof of the modified claim is essentially identical to the proof above, and the stronger results follow immediately.

5. Agreement algorithms with high resiliency. Bracha [5] has devised a beautiful "boot-strapping" method for increasing the resiliency of probabilistic agreement algorithms while also increasing the expected running time. For any constant $\delta > 0$, he shows that n processors can be organized into $m = n^2$ committees, each of size $s = O(\log n)$, such that if fewer than $n/(3+\delta)$ processors are faulty, then at most $\sqrt{m}$ committees are faulty, where a committee is faulty if at least $s/3$ processors in the committee are faulty. Given this structure, each committee simulates a single processor in Ben-Or's protocol. Since this protocol runs in constant expected time when the number of faulty processors is $O(\sqrt{n})$, the running time is proportional to the maximum time needed for each committee to simulate one step of Ben-Or's original protocol.

Processors within a committee decide which messages to send by running Byzantine agreement internally on the messages they have received from other committees and responding accordingly. In addition they must flip a coin visible to the entire committee. Bracha uses any standard Byzantine agreement protocol for the first problem and the coin flipping protocol of Yao [26] (see [8]) or Awerbuch et al. [1] for the second. Thus, the expected running time of Bracha's protocol is $O(s) = O(\log n)$. Using our $\Omega(n/\log n)$-resilient agreement algorithms, we can allow up to $\Omega(m/\log m)$ committees to be faulty, so we might hope that smaller committees would suffice. Although the counting argument in [5] does not yield committees smaller than $\Theta(\log n)$, smaller committees can be formed by applying Bracha's result iteratively. For example, to obtain committees of size $O(\log \log n)$ in a system of n processors, apply the result to obtain n^2 "big committees" of size $O(\log n)$, and then apply the result to each big committee to obtain $m = O(n^2 \log^2 n)$ committees of size $O(\log \log n)$. It is not hard to see that if fewer than $n/(3+\delta)$ processors are faulty then $O(m/\log m)$ committees are faulty. We do not give the details, since the existence of such committee organizations is a special case of the following result, which is proved by a counting argument similar to the one used by Bracha.

LEMMA 5.1. *Let $f(n)$ be a nondecreasing function satisfying $f(nf(n)) = O(f(n))$, and let δ and d be constants with $0 < \delta < 1$ and $d > 1$. For all sufficiently large n and all $t \leqq n/(d+\delta)$, there is a way of organizing n processors into $m \leqq nf(n)$ committees, each of size $s = O((\log f(n))/\log (n/t))$, such that if at most t processors are faulty then at most $m/f(m)$ committees are faulty, where a faulty committee is one that contains at least s/d faulty processors.*

Proof. A *committee* is a set of exactly s processors (s is chosen below). Let $B = \binom{n}{s}$ be the total number of possible choices of committees. For the rest of the definitions in this paragraph, fix a particular set of t faulty processors. A particular committee is *faulty* if it contains at least s/d faulty processors. Let F be the number of faulty committees. F does not depend on the particular choice of the t faulty processors. Let $m = \lfloor nf(n) \rfloor$. An *organization* is a set of exactly m committees. An organization is *bad* if at least

$$h = \lfloor m/f(m) \rfloor + 1$$

committees in the organization are faulty.

To prove the lemma, we want to show that the total number of organizations, $\binom{B}{m}$, exceeds the number of bad organizations. To obtain an upper bound on the number of bad organizations, note that to form a bad organization we must first choose a set of t faulty processors, and then form the organization by choosing h committees from the set of faulty committees and $m-h$ committees from the set of all committees (this overcounts the number of bad organizations). Therefore, it is sufficient to show that the following number ξ is greater than 1:

$$\xi = \binom{B}{m} \Big/ \binom{n}{t}\binom{F}{h}\binom{B}{m-h}. \tag{1}$$

In the following calculations, we let c, c_1, c_2, etc., denote positive constants whose exact values are unimportant. We begin by noting that

$$\binom{B}{m} \Big/ \binom{B}{m-h} = \frac{(B-m+h)(B-m+h-1)\cdots(B-m+1)}{m(m-1)\cdots(m-h+1)}$$

$$\geqq \left(\frac{B-m}{m}\right)^h \geqq \left(\frac{B}{2m}\right)^h.$$

In the last inequality we use $B \geqq 2m$. Noting also that

$$\binom{n}{t} \leqq 2^n \quad \text{and} \quad \binom{F}{h} \leqq \frac{F^h}{h!}$$

and substituting these three bounds in (1), we obtain

$$\xi \geqq \left(\frac{B}{2m}\right)^h 2^{-n} F^{-h} h!. \tag{2}$$

Raising both sides of (2) to the $1/h$ power and noting that $(h!)^{1/h} \geqq h/e \geqq m/(ef(m))$, we have

$$\xi^{1/h} \geqq c_1 B 2^{-n/h} F^{-1} f(m)^{-1}. \tag{3}$$

We now bound the quantities B and F:

$$B = \binom{n}{s} \geqq \frac{(n-s)^s}{s!} = \frac{n^s}{s!}\left(1-\frac{s}{n}\right)^s \geqq \frac{n^s}{s!}\left(1-\frac{s^2}{n}\right),$$

assuming $s \leqq n$. Recall that F is the number of faulty committees (with respect to a particular set of t faulty processors). We form a faulty committee by choosing k of the t faulty processors and $s-k$ of the $n-t$ nonfaulty processors, for some $k \geqq s/d$. Therefore,

$$\begin{aligned} F &= \sum_{s/d \leqq k \leqq s} \binom{t}{k}\binom{n-t}{s-k} \\ &\leqq \sum_{s/d \leqq k \leqq s} \frac{t^k}{k!}\frac{(n-t)^{s-k}}{(s-k)!} \\ &= \frac{n^s}{s!} \sum_{s/d \leqq k \leqq s} \binom{s}{k}\left(\frac{t}{n}\right)^k \left(1-\frac{t}{n}\right)^{s-k} \\ &\leqq \frac{n^s}{s!} 2^{-cs \log(n/t)}. \end{aligned}$$

Only the final inequality requires further explanation. There are two cases. In the first case, $n/t \leqq 4^d$. The sum is the probability that, in s Bernoulli trials with probability of success $t/n \leqq 1/(d+\delta)$ at each trial, the total fraction of successes exceeds $1/d$, and by the Chernoff bound this probability is bounded above by 2^{-bs} for some constant $b > 0$. Since $\log(n/t) \leqq 2d$, we can take $c = b/2d$. In the second case, $n/t > 4^d$ implies $2^d < \sqrt{n/t}$, and we can use a rougher bound:

$$\sum_{s/d \leqq k \leqq s} \binom{s}{k}\left(\frac{t}{n}\right)^k \left(1-\frac{t}{n}\right)^{s-k} \leqq 2^s \left(\frac{t}{n}\right)^{s/d} = \left(\frac{2^d t}{n}\right)^{s/d} < (\sqrt{t/n})^{s/d} = 2^{-(s/2d)\log(n/t)}.$$

Substituting these bounds for B and F into (3) and taking logarithms, we obtain

$$\log \xi^{1/h} \geqq -c_2 + \log\left(1-\frac{s^2}{n}\right) - \frac{n}{h} + cs \log\left(\frac{n}{t}\right) - \log f(m). \tag{4}$$

Since we are going to choose $s = O(\log n)$, the term $\log(1-s^2/n)$ is bounded below by some (negative) constant. We also have, using the assumption that $f(nf(n)) = O(f(n))$,

$$\frac{n}{h} \leqq \frac{nf(m)}{m} \leqq \frac{nf(nf(n))}{nf(n)-1} = O(1).$$

Therefore, by making s a sufficiently large constant multiple of $(\log f(m))/\log(n/t) = O((\log f(n))/\log(n/t))$, the right-hand side of (4) is positive. This completes the proof that $\xi > 1$. □

This result shares with Bracha's the property that a random organization of processors into committees is good with probability approaching 1 exponentially fast as n increases. Unfortunately, for the choice of f and t in which we are most interested, no explicit construction of these "Bracha assignments" is known.

Remark. Although the condition $f(nf(n)) = O(f(n))$ holds for the function $\log n$, it does not hold for functions of the form n^α where α is a constant. Lemma 5.1 can be made to apply to such functions with $\alpha \leqq \frac{1}{2}$ by changing the condition to $f(n^2) = O(f(n)^2)$ and using $m = nf(n)^2$ committees. Only the final calculation, bounding n/h, is affected.

To obtain the next result, we take $d = 4$, $f(n) = 7 \ln n$, and use a variant of Yao's coin flipping protocol within a committee where modifications are needed to ensure that the coins of faulty committees are independent of the coins of nonfaulty committees. This proof of independence does not arise in [5] since the protocol there is based on Ben-Or's $\sqrt{n}$-resilient protocol, which does not require independence.

Let $m = \lfloor 7n \ln n \rfloor$ be the number of committees. We can assume that the size of each committee is $s = 4l+1$ for some integer l. For each committee of size $4l+1$, we form subcommittees consisting of all possible subsets of size $3l+1$. Since a set S of size $3l+1$ could be a subset of several different committees and since S will play a different role in each of these committees, to avoid confusion among these roles we denote a particular subcommittee by the pair (S, C) where S is a subcommittee of the committee C.

ALGORITHM COMMITTEE COIN

Algorithm for processor p_i:

1. For each subcommittee (S, C) with $p_i \in S$ choose and send a value in the following way:
   - 1.1. If p_i is the smallest indexed member in S, randomly choose an integer value v_i uniformly between 1 and m^4 and choose a random bit b_i and broadcast the pair $(v_i, b_i)_{S,C}$ to all members of S.
   - 1.2. Run Byzantine agreement within S on the pair broadcast for (S, C). Let $(v, b)_{S,C}$ denote the resulting pair.
   - 1.3. Using the simultaneous broadcast with erasing primitive (see § 3.2), broadcast $(v, b)_{S,C}$ to the entire network.
2. Attempt to compute a pair $(v, b)_C$ for each committee C:
   - 2.1. For each subcommittee S of C, let $M_{S,C}$ be the multiset of messages $(v, b)_{S,C}$ received from the members of (S, C) as a result of the broadcast in Step 1.3.
   - 2.2. If there is a message that occurs at least $2l+1$ times in $M_{S,C}$, then let $(v, b)_{S,C}$ be this message; otherwise, $(v, b)_{S,C}$ is null.
   - 2.3. If all subcommittees (S, C) of C have been assigned nonnull pairs in Step 2.2, then compute the sum, over all subcommittees of C, of the v values (modulo m^4) and compute the parity of the b values to obtain $(v, b)_C$. Otherwise, no pair $(v, b)_C$ was received from C.
3. Compute the value of the coin as in the Algorithm COIN, viewing each committee as a processor as described above in the outline of Bracha's algorithm.

THEOREM 5.2. *Let $\delta > 0$ be a constant and let $t \leqq n/(4+\delta)$. There is a probabilistic Byzantine agreement algorithm with expected running time of $O((\log \log n)/\log (n/t))$ rounds that is t-resilient to any rushing without eavesdropping adversary.*

Proof. Given the outline of Bracha's result and Lemma 5.1, all that must be shown is that Algorithm COMMITTEE COIN can be used to simulate Steps 1 and 2 of

Algorithm COIN for all committees. More precisely, it is necessary to show (1) that a nonfaulty committee broadcasts the same randomly selected pair to all processors, and (2) that the pairs sent by faulty committees are chosen independently of the pairs of correct committees, although any such pair from a faulty committee can be erased (but not changed) by the adversary after seeing the nonfaulty committees' pairs. Here and subsequently, a faulty processor is one that is made faulty by the adversary at some round during the algorithm.

Before addressing (1) and (2), we note a fact that is useful in proving both. Fix a committee C (either faulty or nonfaulty). By the property of the broadcast primitive, we can imagine, for every subcommittee (S, C) and every processor $p \in S$, that p chooses a message $m_{p,S,C}$ to be broadcast in Step 1.3. (After these choices are made, all the adversary can do later is to erase some of the $m_{p,S,C}$ in the case that p is faulty.) We claim that these choices determine at most one message m_C such that any correct processor either computes $(v, b)_C = m_C$ in Step 2.3 or concludes that $(v, b)_C$ is null. To see this, let $\bar{M}_{S,C}$ be the multiset of messages $\{m_{p,S,C} \mid p \in S\}$ (only messages of the correct form (v, b), for some value v and some bit b, are included). If some message occurs $2l+1$ times in $\bar{M}_{S,C}$ (there is at most one such message since S contains $3l+1$ processors), then let $m_{S,C}$ be this message; otherwise, $m_{S,C}$ is null. If $m_{S,C}$ is nonnull for all subcommittees (S, C), then let m_C be the modular sum of these messages, computed as in Step 2.3. Let q be an arbitrary correct processor. Since the broadcast primitive only allows erasure, q computes $M_{S,C} \subseteq \bar{M}_{S,C}$ in Step 2.1. Therefore, q computes either $(v, b)_{S,C} = m_{S,C}$ or $(v, b)_{S,C} = \text{null}$ in Step 2.2, and finally, q computes either $(v, b)_C = m_C$ or $(v, b)_C = \text{null}$ in Step 2.3.

Consider now a nonfaulty committee C. We first argue that every correct processor computes the same nonnull pair $(v, b)_C$ in Step 2.3. Since C is nonfaulty, there are no more than l faulty processors in it, and so each subcommittee (S, C) contains $3l+1$ members, no more than l of which are faulty. Thus, agreement will be reached in Step 1.2 for any subcommittee (S, C), and $2l+1$ correct processors in (S, C) will send the same pair $(v, b)_{S,C}$ in Step 1.3. Therefore, every correct processor will recover this pair $(v, b)_{S,C}$ in Step 2.2 for every subcommittee (S, C), so every correct processor will compute the same $(v, b)_C$ in Step 2.3. To argue that this $(v, b)_C$ is truly random, note that there is a subcommittee (U, C) that is *pure*, in that U contains no faulty processors. This subcommittee must send a randomly selected pair. The communication within the pure subcommittee cannot be overheard by the adversary, so at the start of Step 1.3, the messages chosen by faulty processors are independent of the pure subcommittee's pair $(v, b)_{U,C}$. It follows that the value v_C (for example) computed in Step 2.3 has the form

$$v_C = v_{U,C} + v_2 + v_3 + \cdots + v_k \qquad (\text{mod } m^4),$$

where k is the number of subcommittees (S, C), where $v_{U,C}$ is distributed uniformly, and where the distribution of $(v_2, \cdots, v_k)$ is unknown (possibly chosen by the adversary) but independent of the value of $v_{U,C}$. Under these conditions, it is clear that v_C is uniformly distributed.

Now consider a faulty committee C (such a committee could consist entirely of faulty processors). At the beginning of Step 1.3, the faulty processors have no information about the randomly chosen pair of any pure subcommittee of any nonfaulty committee. Thus, faulty processors choose messages to send (and possibly erase) independent of these pure subcommittees' pairs, and therefore, of the nonfaulty committees' pairs. As argued above, these choices "lock in" at most one pair m_C such that all the adversary can do after seeing the nonfaulty committees' pairs is to selectively

permit the correct processors to compute $(v, b)_C = m_C$ or $(v, b)_C = \text{null}$ (erasure) in Step 2.3. □

The following immediate corollary of Theorem 5.2 improves the resiliency of Theorem 3.3 at the cost of a more complicated and nonconstructive algorithm.

COROLLARY 5.3. *Let $k > 0$ be a constant, and let $t \leqq \min\{n/(\log n)^k, (n-1)/3\}$. There is a probabilistic Byzantine agreement algorithm with expected running time of $O(1)$ rounds that is t-resilient to any rushing without eavesdropping adversary.*

6. Conclusion. The main contribution of this paper is the simple $cn/\log n$-resilient coin flipping protocol COIN. In order to make the protocol work in the presence of stronger adversaries and apply it to Byzantine agreement, we had to develop other tools. These include (1) a simple 2-round noncryptographic protocol for simultaneous broadcast with erasing; (2) a new combinatorial result that allows the size of Bracha committees to be much smaller than $\Theta(\log n)$; and (3) a combination of probabilistic encryption and secret sharing that is used in ENCRYPTED COIN to foil the blocker.

Acknowledgments. We thank Benny Chor and Andrei Broder for helpful conversations.

REFERENCES

[1] B. AWERBUCH, M. BLUM, B. CHOR, S. GOLDWASSER, AND S. MICALI, *How to implement Bracha's $O(\log n)$ Byzantine agreement algorithm*, Massachusetts Institute of Technology, Cambridge, MA, unpublished manuscript, 1985.

[2] M. BEN-OR, *Another advantage of free choice: Completely asynchronous agreement protocols*, in Proc. 2nd Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1983, pp. 27–30.

[3] M. BEN-OR AND N. LINIAL, *Collective coin flipping, robust voting schemes and minima of Banzhaf values*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 408–416.

[4] M. BLUM, *Coin flipping by telephone*, in Proc. IEEE Spring Comp-Con, IEEE Computer Society, Washington, DC, 1982, pp. 133–137.

[5] G. BRACHA, *An $O(\log n)$ expected rounds randomized Byzantine generals protocol*, J. Assoc. Comput. Mach., 34 (1987), pp. 910–920.

[6] A. Z. BRODER, *A provably secure polynomial approximation scheme for the distributed lottery problem*, in Proc. 4th Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1985, pp. 136–148.

[7] A. Z. BRODER AND D. DOLEV, *Flipping coins in many pockets (Byzantine agreement on uniformly random values)*, in Proc. 25th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1984, pp. 157–170.

[8] B. CHOR AND C. DWORK, *Randomization in Byzantine agreement*, in Advances in Computing Research, Vol. 5, S. Micali, ed., JAI Press, Greenwich, CT.

[9] B. CHOR, S. GOLDWASSER, S. MICALI, AND B. AWERBUCH, *Verifiable secret sharing and achieving simultaneity in the presence of faults*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 383–395.

[10] B. CHOR, M. MERRITT, AND D. B. SHMOYS, *Simple constant-time consensus protocols in realistic failure models*, J. Assoc. Comput. Mach., 36 (1989), pp. 591–614.

[11] R. DEMILLO, N. LYNCH, AND M. MERRITT, *Cryptographic protocols*, in Proc. 14th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1982, pp. 383–400.

[12] D. DOLEV, *The Byzantine generals strike again*, J. Algorithms, 3 (1982), pp. 14–30.

[13] D. DOLEV AND H. R. STRONG, *Authenticated algorithms for Byzantine agreement*, SIAM J. Comput., 12 (1983), pp. 656–666.

[14] C. DWORK, D. SHMOYS, AND L. STOCKMEYER, *Flipping persuasively in constant expected time*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986, pp. 222–232.

[15] P. N. FELDMAN, *Optimal algorithms for Byzantine agreement*, Ph.D. thesis, Department of Mathematics, Massachusetts Institute of Technology, Cambridge, MA, 1988.
[16] ———, personal communication, 1988.
[17] P. FELDMAN AND S. MICALI, *Byzantine agreement in constant expected time (and trusting no one)*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1985, pp. 267-276.
[18] ———, *Optimal algorithms for Byzantine agreement*, in Proc. 20th ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 148-161.
[19] W. FELLER, *An Introduction to Probability Theory and Its Applications*, Vol. I, 3rd ed., John Wiley, New York, 1968.
[20] M. J. FISCHER AND N. A. LYNCH, *A lower bound for the time to assure interactive consistency*, Inform. Process. Lett., 14 (1982), pp. 183-186.
[21] M. J. FISCHER, N. A. LYNCH, AND M. S. PATERSON, *Impossibility of distributed consensus with one faulty process*, J. Assoc. Comput. Mach., 32 (1985), pp. 374-382.
[22] S. GOLDWASSER AND S. MICALI, *Probabilistic encryption*, J. Comput. System Sci., 28 (1984), pp. 270-299.
[23] M. PEASE, R. SHOSTAK, AND L. LAMPORT, *Reaching agreement in the presence of faults*, J. Assoc. Comput. Mach., 27 (1980), pp. 228-234.
[24] M. O. RABIN, *Randomized Byzantine generals*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1983, pp. 403-409.
[25] A. SHAMIR, *How to share a secret*, Comm. ACM, 22 (1979), pp. 612-613.
[26] A. C. YAO, personal communication, 1985.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 500-510, June 1990

008

RESET SEQUENCES FOR MONOTONIC AUTOMATA*

DAVID EPPSTEIN†

Abstract. Natarajan reduced the problem of designing a certain type of mechanical parts orienter to that of finding reset sequences for monotonic deterministic finite automata. He gave algorithms that in polynomial time either find such sequences or prove that no such sequence exists. In this paper a new algorithm based on breadth-first search is presented that runs in faster asymptotic time than Natarajan's algorithms, and in addition finds the shortest possible reset sequence if such a sequence exists. Tight bounds on the length of the minimum reset sequence are given. The time and space bounds of another algorithm given by Natarajan are further improved. That algorithm finds reset sequences for arbitrary deterministic finite automata when all states are initially possible.

Key words. finite automata, reset sequences, breadth first search, automated design, parts orienters

AMS(MOS) subject classification. 68Q20

1. Introduction. Natarajan [5] has considered the design of automated parts orienters; that is, devices that accept mechanical parts in any orientation or in a wide class of orientations, and output them in some predetermined orientation. One such orienter is a pan handler, in which the part slides around on a tray as that tray is tilted, turning in a well-defined way when it hits the walls of the tray. These devices had been previously been described in [2] and [4].

For a given tray and object, and for a given set of possible initial orientations for the object, we desire to determine whether there exists a sequence of tilt angles that will cause the object always to end up in the same orientation. Natarajan made the assumptions that the set of angles is finite, that the set of orientations in which the part can rest on a tray face is also finite, that tilting the tray with a given angle and with the object in a given initial orientation always results in the same final orientation, and that this relation between angles, initial orientations, and final orientations is known. He also made the assumption that all faces of the tray are identical; that is, that the shape of the tray is a regular polyhedron. We will show later how to remove this last assumption. With these assumptions he reduced the problem to the following combinatorial one.

Let $S = \{s_1, s_2, \cdots, s_n\}$ be a set of states, corresponding to orientations of the part to be oriented. Let $\Sigma = \{\sigma_1, \sigma_2, \cdots, \sigma_k\}$ be an alphabet, corresponding to the possible ways of tilting the orienter tray. Finally, let $\delta(\sigma, s)$ be a function indicating the orientation resulting when tilt σ is applied to state s. Then (S, Σ, δ) forms a deterministic finite automaton (DFA). We are further given a set of initial states, or orientations, $X \subset S$. Because the tray is assumed to be symmetric, nothing in the states or transition functions need identify the edge of the tray at which the object is resting.

In what follows, sequences of input symbols to the automaton will be denoted using the letter τ. As with the input symbols themselves, we let $\delta(\tau, s)$ denote the effect of sequence τ on the states of the automaton. Thus, if $\tau = \tau_1\tau_2$, then $\delta(\tau, s) = \delta(\tau_2, \delta(\tau_1, s))$. If τ is the empty input sequence, $\delta(\tau, s) = s$. We denote the set of all

* Received by the editors September 19, 1988; accepted for publication September 19, 1989. This research was performed while the author was a student at the Computer Science Department, Columbia University, New York. It was supported in part by a National Science Foundation student fellowship, by National Science Foundation grants DCR-85-11713 and CCR-86-05353, and by Defense Advanced Research Project Agency (DARPA) contract N00039-84-C-0165.

† Xerox Palo Alto Research Center, 3333 Coyote Hill Road, Palo Alto, California 94304.

possible input sequences by Σ^*. Furthermore, if $X \subset S$, we let $\delta(\tau, X)$ stand as a shorthand for $\{\delta(\tau, x) \mid x \in X\}$. Finally, $\delta^{-1}(\tau, X)$ will be taken to mean $\{s \in S \mid \delta(\tau, s) \in X\}$.

Suppose we are given a set of states, or orientations, $X \subset S$; it is assumed that the initial orientation of the object is one of the states in X. For instance, without assumption we can let $X = S$, because S encodes all the possible orientations of the object. Then the pan-handler orientation problem becomes simply that of finding a sequence $\tau \in \Sigma^*$ such that $|\delta(\tau, X)| = 1$; that is, such that the application of τ will leave the automaton in one particular state no matter at which state in X it began. We call τ a *reset sequence* for (S, Σ, δ) and X.

Alternately, the problem can be phrased in terms of finite functions, instead of automata. In this formulation, we are given a collection of functions from a finite set S to itself, and the problem becomes finding a composition of those functions that takes the subset X to some constant. If $X = S$, the problem becomes that of composing the functions to get a constant function.

Natarajan [5] has given an algorithm for solving this problem for any automaton when $X = S$. This algorithm takes $O(kn^4)$ time, and either produces a reset sequence or proves no such sequence exists. The sequence produced is not guaranteed to be the shortest possible, but Natarajan bounds its length by $O(kn^3)$.

It turns out that for general automata and general X, finding a reset sequence is PSPACE-complete [5]. However Natarajan observes that the automata arising in the pan-handler problem have a property that he calls monotonicity, and that with this property the problem becomes solvable in polynomial time. He gives algorithms with asymptotic time complexity $O(kn^4)$ (or $O(kn^3 \log n)$ when $X = S$), which find sequences of length at most $O(kn^3)$ (respectively, $O(kn^2 \log n)$). The sequences found are again not guaranteed to be optimal.

1.1. New results. This paper presents a new algorithm for finding reset sequences on monotonic automata, for any $X \subseteq S$, which takes time $O(kn^2)$ and is guaranteed to find the shortest possible sequence. Furthermore, this leads to tight worst-case bounds of $n^2 - 2n + 1$ on the number of input symbols in the optimal reset sequence. The algorithm works by defining a new automaton, the states of which correspond to intervals in the cyclic order of the original automaton's states. Reset sequences in the original automaton correspond to paths in the new automaton leading to a singleton interval. Therefore we can find our desired sequence using a simple breadth-first search technique.

As another result of this paper, we extend our technique to certain classes of nonmonotonic automata. In particular, we can use this result to solve the pan-handler problem optimally for asymmetric as well as symmetric pans. The search takes time $O(k^2n^2)$, and the resulting sequence has length at most $kn^2 - 2kn + 1$.

Finally, we improve Natarajan's algorithm for any nonmonotonic automaton and for $X = S$, to take time $O(n^3 + kn^2)$, and working space bounded by $O(n^2)$. We also prove a tighter bound of $O(n^3)$ on the length of the resulting sequence, and show that finding the minimum length reset sequence is NP-complete.

2. Definitions and lemmas. First, we define monotonic automata. Assume that the states of a given deterministic finite automaton (S, Σ, δ) are arranged in some known cyclic order $s_1, s_2, \cdots, s_n$. The transition function δ is *monotonic* if it preserves the cyclic order of the states. Formally, for any input symbol σ, the sequence of states $\delta(\sigma, s_1), \delta(\sigma, s_2), \cdots, \delta(\sigma, s_n)$, after removal of possible adjacent duplicate states, must be a subsequence of a cyclic permutation of $s_1, s_2, \cdots, s_n$. Clearly, if δ is

monotonic, all compositions $\delta(\tau, s)$ will also preserve the cyclic order of the states. From now on in this section, when we refer to the automaton $(S, \Sigma\ \delta)$, we will assume that it is monotonic.

Next let us define an interval $[s_i, s_j]$. This consists of all those states between s_i and s_j (inclusive) in the cyclic order of the states, e.g., $[s_1, s_3] = \{s_1, s_2, s_3\}$. Note that there are n different ways of representing the full set of states S as an interval $[s_i, s_{i-1}]$; any other set of states that can be represented as an interval has exactly one such representation. We say that an interval $J = [s_h, s_i]$ is contained in another interval $I = [s_g, s_j]$, and write $J < I$, when the endpoints of the intervals appear in the cyclic order s_g, s_h, s_i, s_j. Containment as an interval implies containment as a set of states, but the reverse may be false in the case that the containing interval is all of S.

LEMMA 1. *For all $\tau \in \Sigma^*$, and for any interval I, $\delta^{-1}(\tau, I)$ is an interval.*

Proof. If not, there would be $s_{i_1}, s_{i_2}, s_{i_3}$, and s_{i_4} in cyclic order such that $\delta(\tau, s_{i_1})$ and $\delta(\tau, s_{i_3})$ are in I but $\delta(\tau, s_{i_2})$ and $\delta(\tau, s_{i_4})$ are not; but this violates monotonicity. □

Unlike their inverses, the transition functions of the DFA do not necessarily take intervals to intervals. However, we can define a new transition function, corresponding to the original one, that does take intervals to intervals. Let $S' = \{[s_i, s_j] \mid s_i, s_j \in S\} \cup \{\infty\}$, i.e., S' consists of all the possible intervals together with the extra symbol ∞. Let $\alpha \in \Sigma$ and $I = [s_i, s_j] \in S'$, and define $d_x = \delta(\sigma, s_x)$. Then we define δ' as follows. (1) If $d_i \neq d_j$, let $\delta'(\sigma, I) = [d_i, d_j]$. (2) If $d_i = d_{i+1} = d_{i+2} = \cdots = d_j$, let $\delta'(\sigma, I) = [d_i, d_i]$. (3) Otherwise, or if $I = \infty$, let $\delta'(\sigma, I) = \infty$.

The new transition functions we have defined give us a new DFA (S', Σ, δ') whose states are the intervals of the original automaton, and which takes the same input alphabet as the original automaton. Note that this DFA, which is of size $O(kn^2)$, can be constructed in time linear in its size. The only complication is how to determine whether the result of a transition in which the two endpoints are mapped to a single point should be that point or ∞. This can be done by first constructing for each $\sigma \in \Sigma$ and $s \in S$ the interval $\delta^{-1}(\sigma, s)$, which must exist by Lemma 1. This construction takes time $O(n)$, and there are $O(kn)$ intervals to construct, so all such intervals can be constructed in time $O(kn^2)$. Then if the endpoints of interval I are mapped by σ to the same state s, $\delta'(\sigma, I) \neq \infty$ if and only if $I \subset \delta^{-1}(\sigma, s)$. The containment above should be interpreted as being between sets rather than as the interval containment defined earlier; it can be calculated using a constant number of comparisons between interval endpoints to determine interval containment, together with a test for the special case that $\delta^{-1}(\sigma, s) = S$, which is the only case in which set and interval containment can differ.

If τ is an input sequence $\sigma_i \sigma_j \sigma_k, \cdots$, we define $\delta'(\tau, I)$ to be the corresponding composition of interval transition functions. If any of the individual transitions in $\delta'(\tau, I)$ gives ∞, the result as a whole is also ∞. If τ is the empty input sequence we define $\delta'(\tau, I) = I$. Then the following facts follow by an easy induction. Define $d_x = \delta(\tau, s_x)$. (1) If $d_i \neq d_j$, then $\delta'(\tau, I) = [d_i, d_j]$. (2) If $d_i = d_{i+1} = d_{i+2} = \cdots = d_j$, then either $\delta'(\tau, I) = [d_i, d_i]$ or $\delta'(\tau, I) = \infty$. (3) Otherwise, or if $I = \infty$, then $\delta'(\tau, I) = \infty$. That is, the definitions of δ' from δ for single input symbols also hold as theorems for sequences of input symbols, except that occasionally the result will be ∞ in case (2).

LEMMA 2. *For all τ in Σ^*, and all intervals I, if $\delta'(\tau, I) \neq \infty$, then $\delta(\tau, I) \subset \delta'(\tau, I)$.*

Proof. This follows from the characterization of $\delta'(\tau, I)$ above, together with the assumption of monotonicity. □

LEMMA 3. *For all τ in Σ^*, and all intervals I, if $\delta'(\tau, I) \neq \infty$, then for all intervals $J < I$, $\delta'(\tau, J) \neq \infty$ and $\delta'(\tau, J) < \delta'(\tau, I)$.*

Proof. Let $J = [s_i, s_j]$. If $s_i = s_j$, then $\delta'(\tau, [s_i, s_j])$ is easily seen to satisfy the

conditions of the lemma, so assume the two states are different. If $\delta(\tau, s_i) \neq \delta(\tau, s_j)$, then by monotonicity these two states appear in the correct order within $\delta'(\tau, I)$, and again the lemma is satisfied. The remaining case to check is that $\delta(\tau, s_i) = \delta(\tau, s_j)$, and that there is some state s in $[s_i, s_j]$ such that $\delta(\tau, s) \neq \delta(\tau, s_i)$. But this either violates monotonicity or forces $\delta'(\tau, I)$ to equal ∞. □

LEMMA 4. *For all $\sigma \in \Sigma$ and all intervals J, if I is a representation of $\delta^{-1}(\sigma, J)$ as an interval (which by Lemma 1 must exist), and if I is not all of S, then $\delta'(\sigma, I) \neq \infty$ and $\delta'(\sigma, I) \prec J$.*

Proof. Let $I = [s_i, s_j]$. If $\delta'(\sigma, I)$ were equal to ∞, then $\delta'(\sigma, [s_j, s_i])$ would be a singleton interval by monotonicity. But then $\delta(\sigma, S) = \delta(\sigma, I \cup [s_j, s_i]) = \delta(\sigma, I) \cup \delta(\sigma, [s_j, s_i]) \subset J \cup \{\delta(\sigma, s_j)\} = J$, which contradicts the assumption that $\delta^{-1}(\sigma, J) \neq S$. Therefore $\delta'(\sigma, I) \neq \infty$. By the definition of I, $\delta(\sigma, s_i)$ and $\delta(\sigma, s_j)$ are both in J; by monotonicity, they must appear in the correct cyclic order within that interval. It follows that $\delta'(\sigma, I) \prec J$. □

We now prove the main lemma, which shows the equivalence between reset sequences in the original automaton and paths to a single in the interval automaton.

LEMMA 5. *Given $\tau \in \Sigma^*$, and $X \subseteq S$, then $|\delta(\tau, X)| = 1$ if and only if there is a representation of $\delta^{-1}(\tau, \delta(\tau, X))$ as an interval I such that $\delta'(\tau, I) \neq \infty$ and $|\delta'(\tau, I)| = 1$.*

Proof. If there is some such I then, by Lemma 2, τ must be a reset sequence for X. In the other direction, assume we are given τ and X with $\delta(\tau, X) = \{s\}$. We want to find a representation of $\delta^{-1}(\tau, s)$ as an interval meeting the terms of the lemma. We prove the lemma by induction on the length of τ; as a base case, if τ is the empty input sequence, $|X| = 1$ and the lemma clearly holds. Otherwise, assume $\tau = \sigma\bar{\tau}$ for some $\sigma \in \Sigma$ and $\bar{\tau} \in \Sigma^*$. By the induction hypothesis there is a representation of $\delta^{-1}(\bar{\tau}, s)$ as an interval J with $\delta'(\bar{\tau}, J) = [s, s]$.

First assume $I = \delta^{-1}(\sigma, J)$ is not all of S. By Lemma 4 we see that $\delta'(\tau, I) = [s, s]$, as was to be shown.

The remaining case is that $\delta^{-1}(\sigma, J) = S$. Choose the interval $[s_i, s_j]$ such that s_i and s_j are both in $\delta(\sigma, S)$ (and therefore also in J), and also such that $\delta(\sigma, S) \subset [s_i, s_j] \prec J$. This can be done by taking s_i to be the first member in J that is also in $\sigma(S)$, and s_j to be the last such member.

Now if $s_i = s_j$, then $\delta(\sigma, S) = \{s_i\}$, and any representation of S as an interval I will give us $\delta'(\tau, I)$ a singleton interval, satisfying the lemma. So assume $s_i \neq s_j$. This implies that $\delta^{-1}(\sigma, s_j)$ is not all of S, so by Lemma 1 this set has a unique representation as an interval $[s_h, s_k]$. Using monotonicity and the fact that $\delta(\sigma, S) \subset [s_i, s_j]$, it can be shown that $\delta(\sigma, s_{k+1}) = s_i$. Therefore $\delta'(\sigma, [s_{k+1}, s_k])$ is equal to $[s_i, s_j]$. By Lemma 3 we see that $\delta'(\tau, [s_{k+1}, s_k]) = \delta'(\bar{\tau}, [s_i, s_j]) \subset \delta'(\bar{\tau}, J)$ is a singleton interval. □

3. Reset sequences for monotonic automata.

THEOREM 1. *A minimum length reset sequence for monotonic automaton (S, Σ, δ) and initial states $X = \{s_{i_1}, s_{i_2}, \cdots\}$ can be found in time bounded by $O(kn^2)$.*

Proof. First we construct the automaton (S', Σ, δ') as described above. By Lemma 5, a minimum reset sequence for the original automaton will also be a minimum length path in the new automaton from some interval containing X to a singleton interval, and vice versa. By Lemma 3, we need only consider starting from the minimal intervals containing X, rather than all intervals containing X; these minimal intervals can be found as $[s_{i_j}, s_{i_{j-1}}]$. Finally, the shortest path from one of these intervals to a singleton can be found using the standard breadth-first search algorithm. □

THEOREM 2. *If a minimum length reset sequence for monotonic automaton (S, Σ) and initial states X exists, its length is $\leqq n^2 - 2n + 1$.*

Proof. The path constructed by the breadth-first search in Theorem 1 will visit each state of the constructed automaton at most once, and there are n^2+1 states. But the sequence need involve at most one interval representing all of S, and at most one singleton; furthermore, it need never involve state ∞. Thus there are at least $2n-1$ states not included in the minimum length path. □

THEOREM 3. *For any n, there exists a monotonic automaton (S, Σ, δ) with $|S| = n$, and a set of initial states X, such that the minimum length reset sequence for (S, Σ, δ) and X has length n^2-2n+1.*

Proof. Name the states $s_1, s_2, \cdots, s_n$, in that cyclic order. Let Σ consist of only two input symbols, σ_1 and σ_2. Let the transition function always take s_n to s_1, but let σ_1 take all states s_i other than s_n to s_{i+1}, and let σ_2 take all states s_i other than s_n to themselves. That is,

$$\delta(\sigma_1, s_i) = s_{i+1} \quad \text{for } 1 \leqq i < n,$$
$$\delta(\sigma_2, s_i) = s_i \quad \text{for } 1 \leqq i < n,$$
$$\delta(\sigma_i, s_n) = s_1 \quad \text{for } 1 \leqq i \leqq 2.$$

We take $X = S$, so that a reset sequence for X must take all n states to a single state.

Assume we have a reset sequence τ, and define τ_i to be the prefix of τ consisting of the first i symbols of τ. Also define $l(i)$ to be the length of the shortest interval containing all the states in $\delta(\tau_i, S)$. If by $|\tau|$ we denote the number of input symbols in τ, then clearly $l(|\tau|) = 1$. Finally, define $t(j)$, for each j, to be the least i such that $l(i) \leqq j$.

We prove below that, for each $j < n-1$, $t(j) \geqq t(j+1)+n$, that is, there must be at least n input symbols processed between each point at which the shortest interval containing the states becomes shorter. The theorem then follows, because the total number of steps in the reset sequence must be at least $n(n-2)$ for the $n-2$ gaps of n steps each, plus one initial step to reduce $l(t)$ from n to $n-1$.

First note that, if $j \neq 1$, the ith input symbol is σ_1, and $\delta(\tau_i, S) \subset [s_j, s_k]$, then $\delta(\tau_{i-1}, S) \subset [s_{j-1}, s_{k-1}]$. If $j \neq 1$, the ith input symbol is σ_2, and $\delta(\tau_i, S) \subset [s_j, s_k]$, then $\delta(\tau_{i-1}, S \subset [s_j, s_k]$. Therefore, no matter what the input symbols of τ are, if $\delta(\tau_i, S) \subset [s_j, s_k]$, we can see using induction that $\delta(\tau_{i-j-1}, S) \subset I$ for some interval I of length $k-j+1$.

Next observe that if the ith input symbol is σ_1, then $l(i-1) = l(i)$; therefore for each j the input symbol at position $t(j)$ must be σ_2, and furthermore it must be the case that $\delta(\tau_{t(j)-1}, S) \subset [s_n, s_j]$. Using the previous observation we see that $\delta(\tau_{t(j)-n}, S) \subset I$ for some interval I of length $j+1$, and therefore $t(j+1) \leqq t(j)-n$ as was to be proved. The theorem then follows as described above. □

4. Extensions of the monotonic reset sequence technique. Various generalizations of the algorithms and bounds above may be taken. For instance, let us consider the case that what is desired as the result of the reset sequence is a particular state rather than just any single state. The same algorithm as in Theorem 1, but with the breadth-first search terminating only when it reaches the singleton interval corresponding to the desired state, will always find the minimum such reset sequence when it exists, again taking time $O(kn^2)$. The upper bound of Theorem 2 must be relaxed to n^2-n, because it is now possible for the path in the interval automaton to go through all singleton intervals before it gets to the desired one. And the example used in Theorem 3, with the desired singleton state being s_n, requires n^2-n steps for a reset sequence, showing that this new bound is tight.

Another generalization allows us to remove the assumption, used in reducing the pan-handler problem to that of computing reset sequences, that the shape of the

pan-handler is a regular polyhedron. Let us say that the pan-handler has k faces, not necessarily all alike. We assume that the face on which the object is originally resting is known. Then we can form an automaton with kn states, which encode both the position of the object and also the face on which it rests. Again there will be k transition functions; function i will correspond to tilting the pan-handler so that the object moves to face i. For any given tilt of the tray from face i to face j, the change in the orientations of the object will be monotonic; however, because the states also encode the face the object is lying on, the automaton as a whole will not typically be monotonic. Nevertheless, we can use the methods above, building a new automaton the states of which are intervals of object positions together with a single face on which the object rests, and the transition functions of which are the result of applying the original transition functions to these ranges of positions. As above, a breadth-first search through the new automaton results in a reset sequence for the original automaton. The search takes time $O(k^2n^2)$, and the resulting sequence has length at most $kn^2-2kn+1$.

5. Reset sequences for general automata. In this section we will relax the requirement that the automaton (S, Σ, δ) be monotonic, and instead restrict our attention to reset sequences for all of S; that is, we will assume that the automaton may initially be in any of its states, rather than in a state drawn from some subset X of its states. The case we study is easier than the general case because we can never get stuck: if there exists a reset sequence τ, then no matter what sequence $\bar{\tau}$ we have chosen already, $\bar{\tau}\tau$ will still be a reset sequence for the whole set. We can proceed by reducing the size of the set $\delta(\bar{\tau}, S)$ a step at a time, without ever having to worry about backtracking.

The following algorithm, due to Natarajan, works in the above manner to find a reset sequence for any automaton (S, Σ, δ), with the initial set of states being all of S. The reset sequence it finds is not necessarily the shortest possible such sequence. We will put off describing the implementation details of some of the steps until later.

ALGORITHM 1.
begin
    $X \leftarrow S$;
    $\tau \leftarrow$ the empty sequence;
    while $|X| > 1$ **do begin**
        pick $s_i, s_j \in X$ with $s_i \neq s_j$;
        find a sequence $\bar{\tau}$ taking s_i and s_j to the same state;
        $X \leftarrow \delta(bar\tau, X)$;
        $\tau \leftarrow \tau\bar{\tau}$;
    end;
end

THEOREM 4 [5]. *Assuming the steps in the loop of Algorithm* 1 *can be computed, the algorithm terminates after* $O(n)$ *repetitions of the loop, and finds a reset sequence for* (S, Σ, δ) *if such a sequence exists. If the algorithm ever chooses a pair of states* s_i, s_j *such that no sequence* $\bar{\tau}$ *takes the two states to a single state, then no reset sequence exists.*

Proof. Each time through the loop, the size of X decreases by at least one, therefore the loop can be executed at most n times. When the size of X has decreased to one, τ will then be a reset sequence. If any reset sequence τ exists, it will a fortiori satisfy the conditions for $\bar{\tau}$. □

Natarajan described an implementation of the above algorithm that takes time $O(kn^4)$. The two steps of the algorithm that take the most time are finding $\bar{\tau}$ and applying it to X. We now describe some preprocessing that allows these steps to be done quickly, therefore improving Natarajan's result. A naive implementation of these preprocessing steps would take space $O(n^3)$, which is worse than the previous $O(n^2)$

bound; we later show how our preprocessing may be performed within a space bound of $O(n^2)$.

THEOREM 5. *Algorithm* 1 *can be executed in time* $O(n^3+kn^2)$.

Proof. As in the monotonic case, we first form a new automaton of size $O(kn^2)$ and perform a breadth-first search in it. The states of the new automaton consist of each (unordered) pair of states from the original automaton, together with one state for each of the original automaton's states. The result of applying any of the original automaton's input symbols σ to a pair of states (s_i, s_j) will be $(\delta(\sigma, s_i), \delta(\sigma, s_j))$; if $\delta(\sigma, s_i) = \delta(\sigma, s_j)$ then the result will be that singleton state.

Before we run Algorithm 1 itself, we perform a breadth-first search on the new automaton, finding for each pair of original states (s_i, s_j) a shortest input sequence $\tau_{i,j}$ taking that pair to a singleton state. This can be performed in time $O(kn^2)$, and the result can be represented as a shortest-path forest in space $O(n^2)$; paths in this forest lead from each pair to a singleton, along the sequence of pairs found by applying each transition function in $\tau_{i,j}$ successively to the pair (s_i, s_j).

In the following description we will call the above breadth-first search stage 1. If we only desire to know whether there is a reset sequence, without needing to know what that reset sequence is, then we may stop now, having taken time $O(kn^2)$: a reset sequence exists if and only if for every pair (s_i, s_j) such a sequence $\tau_{i,j}$ leading to a singleton can be found.

Next, as stage 2 of our preprocessing, for each pair of states (s_i, s_j) and each state s_k of the original automaton, we compute $\delta(\tau_{i,j}, s_k)$. This is done by performing a pre-order traversal of the shortest path forest computed in stage 1. Whenever we visit a pair (s_i, s_j), we compute in constant time $\delta(\tau_{i,j}, s_k)$, for all states s_k, as follows. Let $\tau_{i,j} = \sigma\tau_{g,h}$, where σ is the first transition function in $\tau_{i,j}$, $\delta(\sigma, s_i) = s_g$, and $\delta(\sigma, s_j) = s_h$. If $s_g = s_h$ let $\tau_{g,h}$ be the empty sequence of transition functions, which corresponds to the identity function. Then $\delta(\tau_{i,j}, s_k) = \delta(\tau_{g,h}, \delta(\sigma, s_k))$ can be computed as one function evaluation of $\delta(\sigma, s_k)$ followed by a table lookup of the value of $\delta(\tau_{g,h}, \delta(\sigma, s_k))$; because we are performing a pre-order traversal the latter value will have already been computed. Since there are $O(n^3)$ computations to be performed, each taking constant time, the total time for this stage is $O(n^3)$.

Now we show how to perform the steps of the main algorithm described above. To find $\bar{\tau}$ for s_i and s_j, we simply look up $\tau_{i,j}$ in the forest we calculated in the first stage; there are $O(n^2)$ pairs, so the shortest sequence $\tau_{i,j}$ resulting in a singleton is at most $O(n^2)$ symbols long, and therefore this step takes time bounded by $O(n^2)$. To find $\delta(\bar{\tau}, X)$ we simply look up, for each member s of X, $\delta(\bar{\tau}, s)$ as calculated in the second stage; $|X| \leqq n$ so this step takes time $O(n)$. The inner loop is executed $O(n)$ times, so the execution of Algorithm 1 as a whole takes time $O(n^3+kn^2)$, which is also the time taken by the preprocessing stages. □

Recall that we claimed that we could reduce the working space used to $O(n^2)$ while keeping the time bounds described above. We do not count the length of the output sequence, for which the best bound we have is $O(n^3)$, as part of this space bound.

The pair automaton we constructed would seem to take $O(kn^2)$ space, but in fact we need only to use constant storage space for each pair of the automaton, and construct the outgoing arcs from each pair as needed from the original automaton. A more serious obstacle to reducing the space is that the space required to store $\delta(\tau_{i,j}, s_k)$ is $\Omega(n^3)$. However, it turns out to be possible to reduce the space required, by keeping $\delta(\tau_{i,j}, s_k)$ only for certain pairs (s_i, s_j) rather than all such pairs. First let us describe an algorithm to compute the pairs for which we will calculate the values of $\tau_{i,j}$. This algorithm is given as input a forest of size x, and another integer parameter y. It

calculates a partition of the forest into $O(x/y)$ subtrees, each of depth at most y.

ALGORITHM 2.
for each vertex v of the forest, in a post-order traversal, **do begin**
  $size(v) \leftarrow 1$;
  for each vertex w such that (w, v) is an edge in the forest **do**
    if $mark(w) = 0$ **then** $size(v) \leftarrow size(v) + size(w)$;
  if $size(v) < y$ **then** $mark(v) \leftarrow 0$;
  else $mark(v) \leftarrow 1$;
end

LEMMA 6. *Algorithm* 2 *takes time linear in* x, *the number of vertices in the forest it processes. After it has been executed, there will be at most* x/y *vertices* v *of the forest with* $mark(v) = 1$. *Furthermore, if we break the outgoing link of each such marked vertex, no tree in the new forest so created will have depth greater than* y.

Proof. The post-order traversal guarantees that $size(w)$ and $mark(w)$ in the inner loop of the algorithm will have been calculated before we process vertex v. For each vertex v, $size(v)$ computes the number of vertices in the subtree of unmarked vertices rooted at v. Each marked vertex has at least $y-1$ unmarked vertices in its subtree, so there can be at most x/y marked vertices. If any tree of unmarked vertices rooted at a marked vertex had depth greater than y, the number of unmarked vertices on any path of length greater than y to the root would be enough to have caused one of the vertices along that path to have been marked; therefore the depth of each such tree is at most y. □

THEOREM 6. *Algorithm* 1 *can be executed in time* $O(n^3 + kn^2)$ *as in Theorem* 5, *using working space bounded by* $O(n^2)$.

Proof. We compute stage 1 as before. But before performing stage 2, we run Algorithm 2 on the shortest path forest computed in stage 1, with x being the number $n(n-1)/2$ of pairs and singletons in the forest, and y equal to n, the number of states in the original automaton.

In stage 2 we now only compute $\delta(\tau_{i,j}, s_k)$ for those pairs (s_i, s_j) that were marked by Algorithm 2. Again we will process each such pair in order by a pre-order traversal of the forest. We first compute the shortest prefix of $\tau_{i,j}$ that takes (s_i, s_j) to another marked pair (s_g, s_h); call this shortest prefix $\bar{\tau}$. By Lemma 6, the number of input symbols in $\bar{\tau}$ is at most n. Then $\delta(\tau_{i,j}, s_k) = \delta(\tau_{g,h}, \delta(\bar{\tau}, s_k))$, which can be computed with at most n function evaluations followed by a table lookup. Using Lemma 6 again we see that there are at most $O(n)$ marked pairs, and for each such pair we have to perform n computations each taking time $O(n)$, so the total time for the new version of stage 2 is again bounded by $O(n^3)$. We store $\delta(\tau_{i,j}, s_k)$ for only $O(n)$ pairs (s_i, s_j), so the total space used is bounded by $O(n^2)$.

In Algorithm 1 itself, the only changed step is in computing $\delta(\tau_{i,j}, X)$. Here (s_i, s_j) might not be marked, but as in stage 2 we can find a shortest prefix $\bar{\tau}$ of $\tau_{i,j}$ taking (s_i, s_j) to a marked pair (s_g, s_h). Again $\bar{\tau}$ has length at most n, so for each member s of X we can find $\delta(\tau_{i,j}, s) = \delta(\tau_{g,h}, \delta(\bar{\tau}, s))$ by $O(n)$ function evaluations followed by a table lookup. The entire computation of $\delta(\tau_{i,j}, X)$ takes time bounded by $O(n^2)$, which does not reduce the running time of the algorithm from that of Theorem 5 by more than a constant factor. □

6. Bounds on the length of reset sequences. In his paper, Natarajan claimed a bound of $O(kn^3)$ on the length of the reset sequences produced by Algorithm 1. This can be tightened as follows.

THEOREM 7. *The reset sequence found by Algorithm* 1, *as implemented in Theorems* 5 *and* 6, *has length at most* $O(n^3)$.

Proof. There are $O(n^2)$ pairs and singletons in the derived automaton, so each $\tau_{i,j}$ has length bounded by $O(n^2)$. The reset sequence as a whole is the concatenation of at most n such sequences, so its length is bounded by $O(n^3)$. □

In fact, the algorithm can be modified so that (s_i, s_j) is always chosen to have the shortest sequence $\tau_{i,j}$ among all pairs remaining in X, within the same asymptotic time and space bounds as before. Using this fact, we can bound the constant factor in the reset sequence length formula above.

If there are x stages left in set X, then those states form $x^2/2+O(x)$ possible pairs; this, combined with the arrangement of all $n^2/2+O(n)$ possible pairs into a breadth-first search tree, shows that the length of $\tau_{i,j}$ can be at most $(n^2-x^2)/2+O(n)$. Therefore, the length of the entire reset sequence will be at most $n^3/3+O(n^2)$.

This bound can be further tightened. We omit detailed proofs of the following facts because it seems unlikely that the bound they give is tight. The key observation is that, as the sequence $\tau_{i,j}$ transforms X, each pair on the path from (s_i, s_j) to a singleton appears first as the image of (s_i, s_j), and not as the image of any other pair: otherwise $\tau_{i,j}$ would not be minimal. As a consequence, the length of $\tau_{i,j}$ can be bounded by $\min((n-x)^2, n(n-x)/2)+O(n)$. This leads to a bound on the length of the entire reset sequence of $11n^3/48+O(n^2)$.

7. Difficulty of computing optimal sequences. The algorithm we described for monotonic automata will always find the shortest possible reset sequence; in contrast, the algorithm for nonmonotonic automata will always find a reset sequence if one exists, but will not necessarily find the shortest such sequence. This raises the question whether it is possible to efficiently find optimum length reset sequences in nonmonotonic automata. We now show that this is unlikely.

THEOREM 8. *Finding the shortest possible reset sequence for an automaton is* NP-*complete.*

Proof. More precisely the problem is, given an automaton and an integer parameter m, to test whether the automaton has a reset sequence of length less than or equal to m. By Theorem 7, such a sequence need have at most polynomial length, so the problem is in NP. We prove completeness by reducing 3-SAT [1] to the problem.

Assume we are given a satisfiability problem as a Boolean formula in conjunctive normal form, with m variables $x_1, x_2, \cdots, x_m$, and with n clauses. The automaton we construct will need only two transition functions σ_1 and σ_2. There will always be a reset sequence of length $m+1$ (in fact any input sequence of that length will reset the automaton), but any reset sequence of length m or less will correspond to a satisfying assignment. The assignment is constructed by letting x_j be true if the jth input symbol of the reset sequence is σ_1, or false if the jth input symbol is σ_2. Conversely, the opposite transformation will produce a reset sequence of length m from any satisfying assignment.

The automaton itself is constructed as follows. It will have one special state r, and $mn+m$ other states $s_{i,j}$ for $1 \leqq i \leqq n$ and $1 \leqq j \leqq m+1$. For all states $s_{i,m+1}$, and for state r, both transition functions will lead to state r. If the ith clause of the formula contains x_j, σ_1 will take state $s_{i,j}$ to r; if that clause contains $\bar{x}_j$, σ_2 will take $s_{i,j}$ to r. We call these transitions to r from states other than $s_{i,m+1}$ *shortcuts.* All remaining transitions take $s_{i,j}$ to $s_{i,j+1}$.

It can be seen that, as we stated above, any input sequence will take all states to r in at most $m+1$ steps. Furthermore, all states except $s_{i,1}$ will always be taken to r in

m steps, so we need only concern ourselves with the former states. If a reset sequence of length m or less exists, then each of these initial states $s_{i,1}$ must be taken by that sequence across a shortcut transition, because otherwise an initial state $s_{i,1}$ would progress through all the states $s_{i,j}$ before reaching r, and that would take $m+1$ steps.

The variable assignment computed from the reset sequence must have a true variable in each clause, corresponding to the shortcut taken by the initial state corresponding to that clause. Thus we see that a satisfying assignment to the formula can be derived from a short reset sequence. Conversely, if an assignment satisfies the formula, the derived input sequence would cause each initial state $s_{i,1}$ to take a shortcut corresponding to the first true variable in the corresponding clause, and we would have a reset sequence of length m. Thus we see that the formula will be satisfiable if and only if the derived automaton has a short reset sequence. □

8. Conclusions and open problems. We have shown that, given a monotonic DFA and a set of initial states it may be in, we can construct a minimum length sequence (if one exists) that takes all the initial states to one particular state. This construction can be performed in time bounded by $O(kn^2)$. Furthermore, we have shown that the length of the resulting sequence is at most n^2-2n+1; there are DFAs for which the minimum reset sequence exists and is this long, so this bound is tight.

We have also shown that, in the general case in which the automaton is not monotonic, we can still find a reset sequence for all of S in time bounded by $O(n^3+kn^2)$ and working space bounded by $O(n^2)$. The length of the resulting sequence is not necessarily optimal, but is bounded by $O(n^3)$.

Some questions remain open. For instance, the algorithm for monotonic automata may be performed in polylogarithmic parallel time using Kučera's breadth-first search algorithm [3], and similarly we may perform the preprocessing in stages 1 and 2 of our algorithm for general automata in *NC*. But the main part of the latter algorithm seems to be inherently sequential: a natural question is whether it too can be performed in parallel, or whether some other algorithm exists that can find reset sequences in parallel. A partial result in this direction is that a reset sequence can be found in random *NC*; this can be done by choosing a long random sequence of pairs of states (s_i, s_j) and concatenating the sequences $\tau_{i,j}$ that take each random pair to a singleton. If there are at least n^3 pairs in the random sequence, then with very high probability the corresponding input sequence will be a reset sequence, and this can be tested in parallel.

Another open problem is the gap between the $O(n^3)$ upper bound on the length of reset sequences for general automata, and the $\Omega(n^2)$ lower bound given for the special case of monotonic automata.

Acknowledgments. I thank Prof. John Kender, who led me to this problem and encouraged me to look for better solutions, and my advisor, Prof. Zvi Galil, for many helpful discussions.

REFERENCES

[1] M. R. Garey and D. S. Johnson, *Computers and Intractibility: A Guide to the Theory of* NP-*Completeness*, W.H. Freeman, San Francisco, 1979.

[2] D. D. Grossman and M. W. Blasgen, *Orienting parts by computer controlled manipulation*, IEEE Trans. Systems Man Cybernet., 5 (1975) pp. 561–565.

[3] L. KUČERA, *Parallel computation and conflicts in memory access*, Inform. Process. Lett., 14 (1982), pp. 93–96.

[4] M. T. MASON AND M. ERDMANN, *An exploration of sensorless manipulation*, in Proc. 3rd IEEE Internat. Conference on Robotics and Automation, San Francisco, CA, April 1986.

[5] B. K. NATARAJAN, *An algorithmic approach to the automated design of parts orienters*, in Proc. 27th Annual Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986, pp. 132–142.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 511-521, June 1990

009

FAST SIMULATIONS OF TIME-BOUNDED ONE-TAPE TURING MACHINES BY SPACE-BOUNDED ONES*

MACIEJ LIŚKIEWICZ† AND KRZYSZTOF LORYŚ†

Abstract. Every single-tape Turing machine (TM) of time complexity $T(n) \geqq n^2$ can be simulated by a single-tape TM in space $T^{1/2}(n)$. It is shown that the time of the simulation can be bounded by $T^{3/2}(n)$ in the case of deterministic TMs and by $T(n)$ in the case of nondeterministic ones. Similar results are shown for off-line machines and for machines with multidimensional tape.

Key words. single-tape TM, off-line TM, time-bounded, space-bounded, time-space tradeoff

AMS(MOS) subject classifications. 3D10, 3D15, 68C40

1. Introduction. One of the most interesting problems in complexity theory is uncovering relationships between time and space complexities of Turing machines (TMs). A problem of this kind is to find for a given function $T(n)$ the minimal function $S(n)$ such that TMs of space complexity $S(n)$ recognize all languages that are recognized by TMs running in time $T(n)$. This problem is still open for most TM models. However some partial results have been obtained, especially for one-tape models: single-tape TMs and off-line TMs. A single-tape TM has a single two-way read-write tape that initially contains the input word. We assume that the tape is infinite to the right only. In an off-line TM the input word is placed on an additional read-only input tape.

In Table 1 we list the smallest known functions $S(n)$ such that TMs of specified type working in space $S(n)$ can simulate any TM of the same type running in time $T(n)$. DTM (NTM) stands for deterministic (nondeterministic, respectively) TM.

TABLE 1

Model	$S(n)$	References
multitape DTM	$T(n)/\log T(n)$	[4], [1]
single-tape DTM and NTM	$T^{1/2}(n)$	[5], [12]
off-line DTM and NTM	$(T(n) \cdot \log n)^{1/2}$	[5], [12]

For one-tape machines even stronger results are known; namely, Paterson [12] has shown that deterministic machines can simulate nondeterministic ones within space $T^{1/2}(n)$. A multidimensional version of Paterson's result, due to Loui [9], says that off-line NTMs with d-dimensional work tape that run in time $T(n)$ can be simulated by off-line DTMs in space $(T(n) \cdot \log T(n))^{d/(d+1)}$.

Of course the space-bounded simulating machines may work much longer than the simulated ones. For example, it is not known if multitape DTMs working in space $T(n)/\log T(n)$ can simulate $T(n)$-time-bounded ones in a time shorter than exponential in $T(n)/\log T(n)$. The situation is quite different in the case of one-tape machines. The best known results are as follows [6], [7]:

* Received by the editors February 29, 1988; accepted for publication (in revised form) August 23, 1989. This research was supported by the Polish government under program no. CPBP 01.01.

† Institute of Computer Science, Wrocław University, 51-151 Wrocław, ul. Przesmyckiego 20, Poland.

(1) Any language accepted by a single-tape DTM in time $T(n) \geqq n^2$ can be accepted by a single-tape DTM in space $T^{1/2}(n)$ and time $T^2(n)$.

(2) Any language accepted by an off-line DTM in time $T(n) \geqq n$ can be accepted by an off-line DTM in space $(T(n) \cdot \log n)^{1/2}$ and time $T^{3/2}(n) \cdot (T^{1/2}(n) + n/(\log n)^{1/2})$.

(3) Any language accepted by a single-tape NTM in time $T(n) \geqq n^2$ can be accepted by a single-tape NTM in space $T^{1/2}(n)$ and time $T^{3/2}(n)$.

(4) Any language accepted by an off-line NTM in time $T(n) \geqq n$ can be accepted by an off-line NTM in space $(T(n) \cdot \log n)^{1/2}$ and time $T(n) \cdot ((T(n) \cdot \log n)^{1/2} + n)/\log n$.

We sharpen the time bounds in all these results. Namely,

(1) For single-tape DTM to $T^{3/2}(n)$,

(2) For off-line DTM to $O(T(n) \cdot ((T(n) \cdot \log n)^{1/2} + n))$,

(3) For single-tape NTM to $T(n)$,

(4) For off-line NTM to $O(T(n) + n \cdot (T(n)/\log n)^{1/2})$.

In the last section we obtain similar results for one-tape multidimensional TMs.

Unless specified differently, the time (space) complexity is used in the following strong sense: a TM has time (space) complexity $F(n)$ if it halts on every input of length n after at most $F(n)$ steps (after visiting at most $F(n)$ tape cells, respectively).

2. Deterministic TMs. Ibarra and Moran [6] constructed an algorithm for simulation of single-tape DTMs of time complexity $T(n)$ by single-tape DTMs running in space $T^{1/2}(n)$ and time $T^2(n)$. We modify this algorithm to reduce the time of simulation to $T^{3/2}(n)$. Then we adapt the algorithm to off-line DTMs.

DEFINITION 1. A sequence $B = (B_0, B_1, B_2, \cdots)$ is a *partition* of a semi-infinite tape F if for each i, B_i consists of a finite number of consecutive cells of F, B_{i+1} is directly to the right of B_i, and each cell of F belongs to a unique B_i. The B_i's are called *tape segments*. If B_0 consists of p cells, and for each $i \geqq 1$, B_i consists of s cells, then B is called $\langle p, s \rangle$ *partition*.

DEFINITION 2. Let M be a single-tape TM, let w be a word in the input alphabet of M, and let $B = (B_0, B_1, B_2, \cdots)$ be a partition of M's tape. *Crossings* of M are pairs (d, q) where $d \in \{-1, 1\}$ and q is a state of M. For each i, the *crossing sequence* of M on input w between B_{i-1} and B_i after t steps, denoted by $\mathrm{CS}(M, w, i, t)$, is a finite sequence of crossings defined as follows:

(a) $\mathrm{CS}(M, w, i, 0)$ is the empty sequence.

(b) $\mathrm{CS}(M, w, i, t) = \mathrm{CS}(M, w, i, t-1)$ if M's head does not cross the boundary between B_{i-1} and B_i during its tth move on input w,

(c) $\mathrm{CS}(M, w, i, t) = \mathrm{CS}(M, w, i, t-1)(d, q)$ if M in the state q moves its head across the boundary between B_{i-1} and B_i in the tth move;

d is -1 if M's head moved left and is 1 if M's head moved right.

DEFINITION 3. Let M, w, B be as in Definition 2. The *history* of M on input w with respect to B after t steps, denoted by $\mathrm{HIST}(M, w, B, t)$, is a finite sequence of crossings and the symbol # defined as follows:

$$\mathrm{HIST}(M, w, B, t) = \mathrm{CS}(M, w, 1, t) \# \mathrm{CS}(M, w, 2, t) \# \cdots \# \mathrm{CS}(M, w, j_t, t)$$

where j_t is the number of the last nonempty crossing sequence. By $|\mathrm{HIST}(M, w, B, t)|$ we denote the number of crossings in $\mathrm{HIST}(M, w, B, t)$.

LEMMA 1. *Let M be a single-tape* TM *of time complexity $T(n)$, let $w \in \Sigma^*$ be a word of length n, and let s be a natural number. Then there is $p \leqq s$ such that for the $\langle p, s \rangle$ partition B we have $|\mathrm{HIST}(M, w, B, T(n))| \leqq T(n)/s$.*

Proof. The conclusion follows from a straightforward application of the Pigeonhole Principle (see [6]). □

DEFINITION 4. $S(n)$ is *fully space constructible* in time $T(n)$ by a single-tape DTM if there is a single-tape DTM, which on any input of length n halts within time $T(n)$ and uses exactly $S(n)$ cells.

THEOREM 1. *Let A be accepted by a single-tape DTM M in time $T(n)$, where $T(n) \geqq n^2$. Assume that $S(n) = T^{1/2}(n)$ is fully space constructible in time $T^{3/2}(n)$ by a single-tape* DTM. *Then A can be accepted by a single-tape* DTM M_1 *in space $S(n)$ and time $T^{3/2}(n)$.*

Proof. We construct a modification of the algorithm given in [6].

Let B_p be the $\langle p, S(n)\rangle$ partition of M's tape, for $p = 1, 2, \cdots, S(n)$. Let w be a word from Σ^*. By Lemma 1, for at least one p_0, $|\mathrm{HIST}(M, w, B_{p_0}, T(n))| \leqq S(n)$. We use the fact that knowing $\mathrm{HIST}(M, w, B_{p_0}, t)$, M_1 can reconstruct the contents of any segment after t steps of M. During the entire simulation M_1 records contents of a single block of the tape, consisting of three segments, and simulates M on this block. Every time M crosses a boundary between segments, M_1 updates the history stored on a separate track of the tape. When M attempts to move its head out of the block, M_1 reconstructs the contents of the new block and resumes simulation on this block.

But M_1 does not know the value of p_0. Therefore it simulates M for a fixed p, starting from $p = 1$, in hope that this p is a good one. However, when it turns out that $\mathrm{HIST}(M, w, B_p, t)$ already contains $S(n)+1$ elements after simulation of t steps, then M_1 seeks the least $p_1 > p$ such that $|\mathrm{HIST}(M, w, B_{p_1}, t)| \leqq S(n)$ and resumes simulation from the step t. This is the main difference between our algorithm and that of [6], since the algorithm of [6] simulates M from the very beginning for each new p.

In the algorithm t denotes the number of M's steps simulated up to a given moment and $\mathrm{HIST}(p, w)$ stands for $\mathrm{HIST}(M, w, B_p, t)$.

ALGORITHM.

Step 1. Construct $S(n) = T^{1/2}(n)$.

Step 2. $p := 1$; $\mathrm{HIST}(p, w) :=$ the empty sequence.

Step 3. (* reconstruction of a block *)
Using $\mathrm{HIST}(p, w)$ recompute the contents of the M's tape block consisting of three segments of B_p: the segment in which the M's head was after t steps and the segments adjacent to it. (In the case when the head is in B_0, the block consists of two segments only: B_0 and B_1.)

Step 4. (* simulation *)
Simulate M on this block from the step t until M reaches a final state or crosses a boundary of segments. If M has reached an accepting (rejecting) state then accept (reject, respectively), otherwise go to Step 5.

Step 5. If $|\mathrm{HIST}(p, w)| = S(n)$, then go to Step 6. Otherwise insert a new element to $\mathrm{HIST}(p, w)$. If M has crossed one of the boundaries of the middle segment of the block then go to Step 4, otherwise (i.e., when M has crossed a block's boundary) go to Step 3.

Step 6. $p_0 := p$.

Step 7. $p := p + 1$.

Step 8. (* creation of a new history *)
Using $\mathrm{HIST}(p_0, w)$ simulate M in succession on all segments, which were visited by M before step t and create $\mathrm{HIST}(p, w)$. If $\mathrm{HIST}(p, w)$ contains more than $S(n)$ elements then go to Step 7, otherwise go to Step 3.

It is easy to see that M_1 uses $O(S(n))$ cells and recognizes the same language as M, so we focus on the time analysis of the algorithm.

Let us note that a single insertion of an element into HIST (p, w) may be done in time $O(T^{1/2}(n))$. This time is also sufficient for finding a single crossing across a boundary of a given segment (in Steps 3 and 8).

We show that the total time of performing each separate step does not exceed $O(T^{3/2}(n))$.

For Steps 1, 2, 6, and 7 this is obvious. Namely, Steps 1 and 2 are executed only once and they need no more time than $T^{3/2}(n)$. Steps 6 and 7 take time $O(\log T(n))$ and by Lemma 1 they are executed at most $T^{1/2}(n)$ times.

Step 3 seeks $O(T^{1/2}(n))$ elements in HIST (p, w) which costs $O(T(n))$, and needs $O(T(n))$ operations of simulation of M. This step is performed once after Step 2, $O(T^{1/2}(n))$ times after Step 8, and after Step 5 whenever M's head crosses a block boundary. Note that whenever M_1 begins the simulation of M on a new block, M's head is at least $T^{1/2}(n)$ cells from the boundaries of the block. So after Step 5, Step 3 may be performed only $O(T^{1/2}(n))$ times. Thus the total cost of Step 3 is $O(T^{3/2}(n))$.

In Step 4, M_1 simulates M step by step, so the total cost of this step is $O(T(n))$.

The total cost of Step 5 is $O(T^{3/2}(n))$. This follows from the fact that $\sum_{1 \leqq p \leqq T^{1/2}(n)} |\text{HIST}(p, w)| \leqq T(n)$.

The cost of Step 8 consists of three components bounded by $O(T(n))$, namely: the cost of simulation, the cost of seeking elements in HIST (p_0, w), and the cost of insertions into HIST (p, w). Since this step can be performed $O(T^{1/2}(n))$ times only, its total cost is $O(T^{3/2}(n))$.

Using the linear speed-up technique of Hartmanis and Stearns [3] we can reduce the complexity of the algorithm to the stated one. □

The algorithm constructed in the proof of Theorem 1 can also be used to simulate off-line DTMs. In this case crossings must contain, apart from a state and a move direction, information about the position of the input head, which causes the algorithm to need more than $T^{1/2}(n)$ space to store a history. On the other hand, $S(n) = (T(n) \cdot \log n)^{1/2}$ space is sufficient, since by Lemma 1 for each input w of length n there is a $\langle p, S(n)\rangle$ partition B such that $|\text{HIST}(M, w, B, T(n))| \leqq (T(n)/\log n)^{1/2}$. Let us note also that a single insertion of one element into history, organized as previously, can cost as much as $S(n) \cdot \log n$ steps, which together with the fact that even $T(n)$ insertions may be needed implies that the algorithm would run for at least $(T(n) \cdot \log n)^{3/2}$ steps. However we can reduce this cost to $T(n) \cdot [(T(n) \cdot \log n)^{1/2} + n]$.

THEOREM 2. *Let A be accepted by an off-line* DTM M *in time $T(n)$, where $T(n) \geqq n$. Let $S(n) = (T(n) \cdot \log n)^{1/2}$ be fully space constructible by an off-line* DTM *in time $T_1(n) = T(n) \cdot [(T(n) \cdot \log n)^{1/2} + n]$. Then A can be accepted by an off-line* DTM M_1 *in space $S(n)$ and time $O(T_1(n))$.*

Proof. The machine M_1 performs the algorithm given in the proof of Theorem 1. However now HIST (p, w) has the following form after the simulation of t steps of M:

$$(*) \qquad \text{CS}(M, w, 1, t_0), \cdots, \text{CS}(M, w, j, t_0), k, (d_1, q_1, l_1), \cdots, (d_r, q_r, l_r),$$

where

—t_0 is the number of M's steps simulated before the last execution of Step 8 by M_1,

—k is the number of the work tape segment, visited by M's head in step t_0,

—(d_i, q_i, l_i) are consecutive crossings between the segments, recorded between the t_0th and tth step (d_i is the direction of move, q_i is the state of M, and l_i is the input head position),

and the numbers k and l_i are written in binary.

Such an organization of HIST (p, w) enables M_1 to insert one crossing into it in $O(S(n)+n)$ steps, so the total cost of all insertions is $O(T_1(n))$.

In order to evaluate the cost of Step 3, we describe it in more detail. Let HIST (p, w) be stored in the form (*). M_1 does the following:

(i) Evaluates $m = k + \sum_{i=1}^{r} d_i$ (i.e., m is the number of the segment currently visited by M's head),

(ii) Marks in HIST (p, w) all crossings through the left boundary of the $(m-1)$st segment and the right boundary of the $(m+1)$st one,

(iii) Simulates M; whenever M attempts to cross the block boundaries, M_1 finds a succeeding crossing marked at (ii), places the input head at the position stored in this crossing, and resumes the simulation.

While executing (i) and (ii), the machine M_1 runs over the tape dragging a binary counter along with its head. Since the counter's length does not exceed $\log T(n)$ both (i) and (ii) can be performed in time $O(S(n) \cdot \log T(n))$. More costly is (iii). The simulation costs $O(T(n))$ and each of the crossing seekings costs $O(S(n)+n)$, which gives $O(T(n)+n \cdot (T(n)/\log n)^{1/2})$ as a cost of (iii). Since Step 3 is executed at most $O((T(n) \cdot \log n)^{1/2})$ times, its total cost is $O(T_1(n))$.

In order to achieve the stated time complexity, we must use a fast method of seeking elements in HIST (p_0, w) in Step 8. In particular, a method of marking the crossings (at the beginning of each segment simulation) faster than $S(n) \cdot \log T(n)$ should be used.

At the beginning of Step 8, M_1 evaluates m as in Step 3, but while doing so divides the part of the tape occupied by $(d_1, q_1, l_1), \cdots, (d_r, q_r, l_r)$ into sectors $t_1, t_2, \cdots, t_s$ of length not greater than $2\lceil \log T(n) \rceil$. Within each sector t_j $(j = 1, \cdots, s)$ M_1 writes down in binary the number $m_j = k + \sum_{i=1}^{u_j} d_i$, where u_j is the number of the last crossing stored in t_j (see Fig. 1).

It is easy to see that this can be done in time $O(S(n) \cdot \log T(n))$.

Let us note that only crossings across the boundaries of the segments of the numbers from $\langle m_{j-1} - v_j, m_{j-1} + v_j \rangle$ may be stored in t_j, where v_j is the number of crossings in t_j. M_1 uses this fact during the marking of crossings in HIST (p_0, w). M_1 simulates M segment by segment and decrements the numbers $m_0, m_1, \cdots, m_{s-1}$ at the end of the simulation of each segment so that each m_j describes the distance of the currently simulated segment from the segment reached by M after the last crossing written in t_j. In order to mark the crossings across the succeeding boundary, M_1 looks through its tape searching for the sectors t_j such that m_{j-1} is from the interval $\langle -v_j, v_j \rangle$. This may be easily done since we can assume that the values v_j $(j = 1, \cdots, s)$ are stored on the third track below m_{j-1}. The values v_j may be evaluated in time $O(S(n) \cdot \log \log T(n))$. If m_{j-1} is from $\langle -v_j, v_j \rangle$, then M_1 marks in t_j all crossings (d_x, q_x, l_x), for which $z_x = m_{j-1} + \sum_{i=1+u_{j-1}}^{x} d_i = 0$. Since $v_j \leqq 2\lceil \log T(n) \rceil$ and at the beginning of each segment simulation all m_j are decremented, for each t_j values z_x are evaluated at most $4 \cdot \lceil \log T(n) \rceil$ times. If M_1 evaluates z_x's dragging the current sum

CS (M, w, j, t_0)		sector t_1	sector t_2	
	k	$(d_1, q_1, l_1) \cdots (d_{u_1}, q_{u_1}, l_{u_1})$	$(d_{u_1+1}, q_{u_1+1}, l_{u_1+1}) \cdots (d_{u_2}, q_{u_2}, l_{u_2})$	
	$m_0 = k$	m_1	m_2	

FIG. 1

along with its head, it is clear that the total cost of computing the z_x's is $O(S(n) \cdot \log T(n) \cdot \log\log T(n)) = O(T(n))$. To the cost of the crossings marking we must add the cost of looking for "good" m_j's, which is $O(T(n))$. Thus $O(T(n))$ is the total cost of crossing marking.

It is easy to check that the cost of simulation of M on consecutive segments is $O(T(n)+n \cdot (T(n)/\log n)^{1/2})$ and it is as well the cost of the whole Step 8, because the cost of all insertions into HIST (p, w) has been already evaluated separately. Since Step 8 may be performed at most $O(S(n))$ times, its total cost is

$$O(S(n) \cdot [T(n)+n \cdot (T(n)/\log n)^{1/2}]) = O(T_1(n)). \qquad \square$$

The same algorithm can be used if $T(n)$ is not fully space constructible. However in this case the time of simulation is a little longer.

COROLLARY 1. (a) *Let A be accepted by a single-tape* DTM M *in time $T(n)$, where $T(n) \geqq n^2$. Then A can be accepted by a single-tape* DTM M_1 *in space $S(n) = T^{1/2}(n)$ and time $T^{3/2}(n) \cdot \log T(n)$.*

(b) *Let A be accepted by an off-line* DTM M *in time $T(n)$, where $T(n) \geqq n$. Then A can be accepted by an off-line* DTM M_1 *in space $S(n) = (T(n) \cdot \log n)^{1/2}$ and time $O(T(n) \cdot [(T(n) \cdot \log n)^{1/2} + n] \cdot \log T(n))$.*

Proof. M_1 does the simulation for $S(n) = 1, 2, 4, 8, \cdots, 2^i, \cdots$. Note that the simulation succeeds for the smallest i such that $2^i \geqq S(n)$. $\square$

3. Nondeterministic TMs. If we want to simulate a single-tape NTM M working in time $T(n)$ by a single-tape NTM M_1 working in space $T^{1/2}(n)$, then using the power of nondeterminism we can construct M_1 so that it also works in time $T(n)$. Machine M_1 guesses a proper $\langle p, T^{1/2}(n)\rangle$ partition B and HIST $(M, w, B, T(n))$ no longer than $T^{1/2}(n)$ and then simulates M on separate segments. It is easy to see that M_1 accepts in $T(n)$ steps all words accepted by M. On the other hand, we can easily check that if M_1 accepts a word w then the guessed history corresponds to some accepting computation of M on w. However M_1 can perform infinite computations even if M is strongly time-bounded. These computations may be easily halted if $T^{1/2}(n)$ is fully space constructible because then M_1 can count simulated steps. Moreover if the counter is always placed near the head, then M_1 works in time $O(T(n) \cdot \log T(n))$. In order to achieve time of simulation $T(n)$, we use a very clever counter constructed by Fürer [2] to refine the time hierarchy for k-tape DTMs ($k \geqq 2$). Although Fürer's counter was constructed for DTMs with at least two tapes, we show that in many cases it can be handled by a one-tape nondeterministic TM. Let us outline the construction from [2].

The counter has the form of a full binary tree in which each node contains a B-ary digit (for some fixed B). Let for each node v, $h(v)$ denote the height of v (i.e., the distance from v to the leaves), and let $d(v)$ denote the digit stored in v. The value of the counter is

$$\sum_{v \in \text{Tree}} d(v) \cdot B^{h(v)}.$$

If we want to decrement the counter, we may subtract one from any of the leaves. If we choose a leaf v already containing zero, then we must find the nearest ancestor u of v with nonzero contents, subtract one from u and put $B-1$ into all nodes lying on the path from v to u. We say that the counter is *exhausted* when we attempt to subtract one from the root that already contains zero.

The machine M_1 stores the counter's nodes in inorder in consecutive cells of its tape (see Fig. 2). On a separate track below each node (except the root) it writes down L or R, depending on whether the node is the left or the right son of its father.

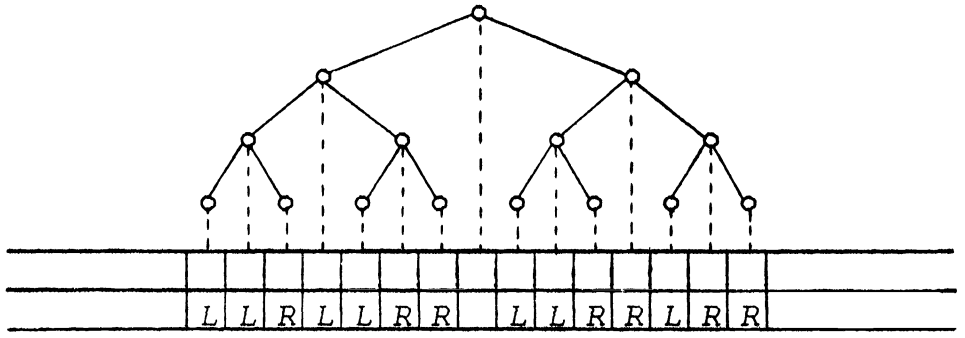


FIG. 2

By length of the counter we mean the number of nodes in the tree. Thus only the numbers of the form $2^i - 1$ for some i may be the lengths of the counters.

LEMMA 2. *Let* $B = 4$.

(a) *The full counter (i.e., with all nodes containing three) of length n represents the number* $(3/2) \cdot (n+1) \cdot n = O(n^2)$.

(b) *Let us suppose that we start with full counter of length n and after m decrements the counter is exhausted. Then* $m \geqq (n+1)^2 - 1$.

(c) *m successive decrements of the initially full counter of length n can be performed by* M_1 *in* $O(m)$ *steps.*

Proof. Parts (a) and (b) are obvious.

For (c), let us note that nodes of height h are 2^{h-1} cells from their sons, and at most m/B^h carries are necessary from nodes of height $h-1$ to nodes of height h. It is easy to see that a single carry can be done in $O(h \cdot 2^{h-1})$ steps, so the total time of m decrements is at most

$$O\left(\sum_{h=0}^{\lceil \log_2(n+1) \rceil - 1} \left(\frac{m}{B^h}\right) \cdot h \cdot 2^{h-1}\right) = O(m). \qquad \square$$

THEOREM 3. *Let A be accepted by a single tape* NTM M *in time* $T(n)$, *where* $T(n) \geqq n^2$. *Let* $T^{1/2}(n)$ *be fully space constructible in time* $T(n)$. *Then A can be accepted by a single-tape* NTM M_1 *in space* $T^{1/2}(n)$ *and time* $T(n)$.

Proof. Let k satisfy the condition $2^k \geqq T^{1/2}(n) + 1 > 2^{k-1}$. For $B = 4$, M_1 constructs the full counter of length $2^k - 1$. Note that this construction can be done in time $k \cdot (2^k - 1)$. Namely, at the beginning M_1 marks a zone consisting of $2^k - 1$ cells and puts 3 (i.e., $B-1$) into each cell of the zone. Then in $k-1$ stages it labels the cells by L or R. During each stage M_1 moves its head through the zone writing down L and R by turns in every other cell. Thus at the ith pass M_1 labels the cells corresponding to the counter's nodes of the height $i-1$.

Then M_1 simulates M on segments of nondeterministically guessed $\langle p, T^{1/2}(n) \rangle$ partition. When the counter is exhausted then M_1 interrupts the simulation and rejects. By Lemma 2(b), it can happen only when M_1 has simulated at least $2^{2k} - 1 \geqq T(n)$ steps, so any accepting computation of M can be successfully simulated. Moreover, by Lemma 2, M_1 works in time $O(T(n))$.

We can decrease the time and space to the stated ones by applying the well-known linear speed-up technique of Hartmanis and Stearns [3]. $\square$

Similarly as in the case of DTMs, an off-line NTM M working in time $T(n)$ can be simulated in space $(T(n) \cdot \log n)^{1/2}$. However this space is inappropriate to use a single counter. The next lemma shows that we can divide the tape of the simulating

machine into blocks and count simulated steps in each block separately, starting with full counters.

LEMMA 3. *Let H be an off-line* NTM. *Then for each finite computation of H there is a partition of the work tape into blocks $B_1, B_2, \cdots$ of consecutive cells such that for each block*

(i) *The length n_j of block B_j is equal to $2^{i_j}-1$ for some i_j;*

(ii) *The number T_j of steps made by H during the computation within B_j satisfies the condition*

$$(\tfrac{1}{4}) \cdot (n_j+1)^2 - 1 < T_j \leqq (n_j+1)^2 - 1.$$

Sketch of proof. Let us assume that H makes T steps during a fixed computation. Let t_k be the number of moves over the kth cell. Let m be the last cell belonging to B_{j-1}. Then we put n_j to be the minimal 2^i-1 such that

$$\sum_{p=1}^{2^i-1} t_{m+p} \leqq 2^{2i} - 1.$$

If necessary, we include some cells (no more than $T^{1/2}$) not visited by H's head in the last block. □

THEOREM 4. *Let A be accepted by an off-line* NTM M *in time $T(n)$, $T(n) \geqq n$. Let $S(n) = (T(n) \cdot \log n)^{1/2}$ be fully space constructible in time $T_1(n) = T(n) + n \cdot (T(n)/\log n)^{1/2}$. Then A can be accepted by an off-line* NTM M_1 *in space $S(n)$ and time $O(T_1(n))$.*

Sketch of proof. M_1 nondeterministically divides the tape of length $(T(n) \cdot \log n)^{1/2}$ into blocks and puts a full counter for $B = 4$ into each block. Then it checks if

$$\sum_{j=1}^{s} \left[\left(\frac{1}{4}\right) \cdot (n_j+1)^2 - 1 \right] \leqq T(n), \tag{1}$$

where s is the number of the blocks and n_j $(j = 1, \cdots, s)$ denotes the length of the jth block. Then M_1 simulates M decrementing a counter after each step of M and rejecting when any of the counters is exhausted. Note that condition (1) is fulfilled if M_1 has guessed a partition as in Lemma 3, so M can be successfully simulated by M_1. Moreover, this condition implies that the total contents of all counters is $O(T(n))$, so M_1 can make at most $O(T_1(n))$ steps. □

As for DTMs we can abandon the assumption of the constructibility of $T(n)$, but then we must multiply $T_1(n)$ by the factor $\log T(n)$ in Theorem 3 and 4.

4. Multidimensional TMs. The algorithm used in the proof of Theorem 1 can also be adapted for simulation of off-line DTMs with multidimensional tape. We describe the modifications required in the case of two-dimensional machines. We show that a two-dimensional DTM M of time complexity $T(n)$ can be simulated by a two-dimensional DTM M_1 working in space $S(n) = (T(n) \cdot \log T(n))^{2/3}$ and time $T_1(n) = O(T^{5/3}(n) \cdot [\log T(n) + n/(T(n) \cdot \log T(n))^{1/3}]/\log^{1/3} T(n))$.

As in [6] by the $\langle p, s \rangle$ partition $(1 \leqq p \leqq s)$ of the two-dimensional tape, we mean the partition in which each segment is a square, which for some u and v contains all cells (x, y) satisfying $us - p \leqq x < (u+1)s - p$, and $vs - p \leqq y < (v+1)s - p$. The numbers u, v are called *segment coordinates*. Lemma 1 has the following analogue (see [6]):

LEMMA 4. *Let a two-dimensional off-line* DTM M *be of time complexity $T(n)$. Let w be in Σ^*, $|w| = n$, and let $s \leqq T(n)$. Then for some $\langle p, s \rangle$ partition B the machine M crosses the boundaries between the segments of B at most $T(n)/s$ times working on w.*

Let us note that the information indicating a position of heads, a state of M, and a direction of move when M crossed a boundary between two segments is not sufficient to reconstruct the contents of segments. This is due to the fact that M_1 cannot set the crossings of history created in Step 8 in chronological order. Such an order is not necessary in the case of one-dimensional tape, since then the segments are bounded by two boundaries and the head can return to a segment across this boundary only, across which it left this segment last time. Now the situation is quite different, the head can return across any of four boundaries.

In order to secure the correctness of the simulation, each crossing additionally contains the number of the step in which this crossing was made. Thus the history is now a sequence of quintuples $\langle d_i, q_i, l_i, h_i, t_i\rangle$, ordered according to t_i's, where q_i, l_i are as in the definition of the history for one-dimensional off-line TM, $d_i \in \{\langle -1, 0\rangle, \langle 1, 0\rangle, \langle 0, -1\rangle, \langle 0, 1\rangle\}$ indicates one of four directions, h_i indicates the exact position of the work head, and t_i is the number of the step in which this crossing was made. Since a single crossing can be stored in $O(\log T(n))$ cells and (by Lemma 4) for some $\langle p, (T(n) \cdot \log T(n))^{1/3}\rangle$ partition, the corresponding history has at most $T^{2/3}(n)/\log^{1/3} T(n) = S(n)/\log T(n)$ crossings, the whole history can be stored in a single segment.

In Step 3, M_1 reconstructs the contents of the block consisting of the segment currently visited by the head and the eight segments adjacent to it. At the beginning of this step M_1 marks each crossing in HIST (p, w), after which M's head entered this block. It is easy to see that this marking can be made in $O(S(n) \cdot \log T(n))$ steps. Then M_1 simulates M in this block. Since the crossings are chronologically ordered the history is, in fact, searched only once during the simulation. Namely, whenever a succeeding crossing is needed M_1 finds in time $O(S^{1/2}(n))$ the position of the crossing that was found last time and starting from this position looks for the next marked crossing. After that, M_1 places its heads according to the information stored in the crossing. Since M can move out of the block at most $S(n)/\log T(n)$ times, the total time needed to resume the simulation is

$$O([S(n)/\log T(n)] \cdot (n + S^{1/2}(n) \cdot \log T(n))) = O(T(n) \cdot [\log T(n) + n/(T(n) \cdot \log T(n))^{1/3}]).$$

This results in $O(T_1(n))$ as the total cost of Step 3, because this step can be performed at most $T^{2/3}(n)/\log^{1/3} T(n)$ times.

In Step 4, M_1 counts the simulated steps. Since M_1 keeps the counter near the head, the total cost of Step 4 is $O(T(n) \cdot \log T(n))$.

More radical modifications are needed in Step 8. The main problems M_1 must solve are as follows:

(i) Before the simulation on each segment, M_1 must mark the proper crossings in a short time,

(ii) M_1 must be able to designate all segments visited till then by M's head,

(iii) After the simulation on all these segments, M_1 must restore the chronological order of the history.

Thus Step 8 now assumes the following shape:

8.1. HIST (p, w) := "the empty sequence."

8.2. Label all crossings in HIST (p_0, w) as "unmarked."

8.3. For each crossing e in HIST (p_0, w) compute the coordinates of the segment that M's head entered when making e. Store these coordinates on the second track below e.

8.4. Simulate M on the segment of the coordinates $(0, 0)$. Whenever M crosses any boundary between segments in the new $\langle p, S^{1/2}(n)\rangle$ partition, insert a new element into HIST(p, w). Note that the parameter t_i in the new crossings can be easily computed. If HIST(p, w) contains more than $T^{2/3}(n)/\log^{1/3} T(n)$ elements (i.e., $S(n)$ cells do not suffice to store HIST(p, w)), then go to Step 7. Label all crossings with the coordinates $(0, 0)$ as "marked." If all crossings in HIST(p_0, w) are already "marked" then go to step 8.6.

8.5. (* Main loop: M_1 visits in turn all segments visited by M and simulates M whenever it visits a segment for the first time *)
For each crossing $\langle d, q, l, h, t\rangle$ from HIST(p_0, w) **do** the following:
(a) Move the coordinate system in the direction d; to this end subtract d from the coordinates stored below each crossing.
(b) If crossings with the coordinates $(0, 0)$ are "unmarked" then **perform** step 8.4.

8.6. Sort HIST(p, w) chronologically.
HIST(p, w) can be treated as a table with $(T(n) \cdot \log T(n))^{1/3}$ rows and $T^{1/3}(n)/\log^{2/3} T(n)$ columns. Each element of this table occupies $\log T(n)$ cells. Note that the cost of a transposition of two adjacent elements is less if they are in the same column than if they are in the same row. This justifies the choice of the following algorithm of table sorting:
(i) Select the smallest element in each column and place them in the first row.
(ii) Select the smallest element in the first row and remove it from the table.
(iii) If the table is nonempty then go to (i).

Now we evaluate the cost of Step 8. Of course steps 8.1 and 8.2 have no effect on this cost. Step 8.3 can be in an obvious manner performed in $O(S(n) \cdot \log T(n))$ steps. The most expensive parts of Step 8 are steps 8.4 and 8.5. We can divide the cost of these steps into the following components:

(a) $O(T(n) \cdot \log T(n))$—the cost of simulation,

(b) $O([S(n)/\log T(n)] \cdot (n + S^{1/2}(n) \cdot \log T(n)))$—the cost of handling the histories,

(c) $O([S(n)/\log T(n)] \cdot S(n))$—the cost of adjusting the coordinates,

which totals

$$O(T^{4/3}(n) \cdot \log^{1/3} T(n) + n \cdot T^{2/3}(n)/\log^{1/3} T(n)).$$

It is obvious that M_1 can find a minimal element in a column as well as in a row in time $O(S^{1/2}(n) \cdot \log T(n))$, so the sorting algorithm at Step 8.6 can be simply implemented to run in time $O(T(n) \cdot \log T(n))$. Thereby the cost of the whole Step 8 is

$$O(T^{4/3}(n) \cdot \log^{1/3} T(n) + n \cdot T^{2/3}(n)/\log^{1/3} T(n)).$$

Since this step can be performed at most $(T(n) \cdot \log T(n))^{1/3}$ times, its total cost is $O(T^{5/3}(n) \cdot \log^{2/3} T(n) + n \cdot T(n)) = O(T_1(n))$.

Clearly, the above considerations generalize to off-line DTMs of any dimension.

THEOREM 5. *Let A be accepted by an off-line* DTM *M with an r-dimensional storage tape in time $T(n) \geqq n$. Let $r' = 1/(r+1)$ and $T_1(n) = T^{2-r'}(n) \cdot (\log T(n) + n/(T(n) \cdot \log T(n))^{r'})/\log^{r'} T(n)$. Let $S(n) = (T(n) \cdot \log T(n))^{r/(r+1)}$ be fully space constructible by a one-dimensional off-line* DTM *in time $T_1(n)$. Then A can be accepted by an off-line* DTM *M_1 with an r-dimensional storage tape in space $S(n)$ and time $O(T_1(n))$.*

5. Conclusions. The problem of efficient simulations between different TM models has been intensively studied by many researchers. Although many efficient simulations have been obtained, it is not known in most cases if they are optimal. Recently, significant progress has been made in the case of one-tape machines; in particular, several lower time-bounds were obtained for simulations on these machines (see, e.g., [8], [10], [11]).

We have presented some fast simulations between one-tape TMs. Although the simulations for one-tape NTMs are optimal it remains open if the time complexity of the simulations can be improved in the case of DTMs.

Acknowledgments. We are very indebted to Professor Leszek Pacholski for his guidance and many valuable discussions. We would also like to thank the referees for numerous corrections and suggestions for improvements.

REFERENCES

[1] L. M. ADLEMAN AND M. C. LOUI, *Space-bounded simulation of multitape Turing machines*, Math. Systems Theory, 14 (1980), pp. 215–222.

[2] M. FÜRER, *The tight deterministic time hierarchy*, in Proc. 14th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1982, pp. 8–16.

[3] J. HARTMANIS AND R. E. STEARNS, *On the computational complexity of algorithms*, Trans. Amer. Math. Soc., 117 (1965), pp. 285–306.

[4] J. E. HOPCROFT, W. J. PAUL, AND L. G. VALIANT, *On time versus space*, J. Assoc. Comput. Mach., 24 (1977), pp. 332–337.

[5] J. E. HOPCROFT AND J. D. ULLMAN, *Relations between time and tape complexities*, J. Assoc. Comput. Mach., 15 (1968), pp. 414–427.

[6] O. H. IBARRA AND S. MORAN, *Some time-space tradeoff results concerning single-tape and offline* TM's, SIAM J. Comput., 12 (1983), pp. 388–394.

[7] S. KURTZ AND W. MAASS, *Some time, space and reversal tradeoffs*, in Conference Computing Complexity Theory, Santa Barbara, March 21–25, 1983, pp. 30–40.

[8] M. LI AND P. M. B. VITANYI, *Tape versus queue and stacks: The lower bounds*, Inform. and Comput., 78 (1988), pp. 56–86.

[9] M. C. LOUI, *A space bound for one-tape multidimensional Turing machines*, Theoret. Comput. Sci., 15 (1981), pp. 311–320.

[10] W. MAASS, *Combinatorial lower bound arguments for deterministic and nondeterministic Turing machines*, Trans. Amer. Math. Soc., 292 (1985), pp. 675–693.

[11] 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 1987, Association for Computing Machinery, New York, 1987, pp. 94–100.

[12] M. S. PATERSON, *Tape bounds for time-bounded Turing machines*, J. Comput. Systems Sci., 6 (1972), pp. 116–124.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 522–537, June 1990

010

UNBOUNDED SEARCHING ALGORITHMS*

RICHARD BEIGEL†

Abstract. The unbounded search problem was posed by Bentley and Yao. It is the problem of finding a key in a linearly ordered unbounded table, with the proviso that the number of comparisons is to be minimized. It is shown that Bentley and Yao's lower bound is essentially optimal, and some new upper bounds for the unbounded search problem are proven. The solution of this problem in parallel is demonstrated.

Key words. unbounded search, table lookup, prefix code, computability, algorithm

AMS(MOS) subject classifications. 68Q99, 68P10

1. Introduction. As posed in [BY76], unbounded search is the problem of searching for a key in a sorted table of unbounded size. The following two-player game is equivalent to the unbounded search problem: Player A chooses an arbitrary positive integer n. Player B is allowed to ask whether an integer x is less than n. In general the number of questions that B has to ask in order to determine n is a function of n. In this paper, we present lower and upper bounds on the size of this function.

The following theorem from [BY76] is instrumental in providing lower bounds on the number of questions needed to solve the unbounded search problem.

THEOREM 1 (Bentley and Yao). *If $f(n)$ questions suffice to solve the unbounded search problem, then $f(n)$ satisfies Kraft's inequality*:

$$\sum_{n \geqq 1} 2^{-f(n)} \leqq 1.$$

Bentley and Yao proved this by noting that if $\sigma(n)$ is the sequence of answers given by A in the two-player game above, then the function σ determines a prefix code for the positive integers.

We write Z^+ to denote the set of positive integers.

Notation 2. If f is a function from Z^+ to Z^+, then

$$\mathrm{Kr}(f) = \sum_{n \geqq 1} 2^{-f(n)}.$$

2. Upper bounds for unbounded search. In this section, we prove a partial converse to Theorem 1.

We say that a real number x between 0 and 1 is *recursive* if there is an algorithm to compute the ith bit of the binary expansion[1] of x. Equivalently, x is recursive if there is a fixed algorithm that takes a rational number y as input and determines if $x < y$. Alternatively, x is recursive if the sequence of bits in the mantissa of x is the characteristic sequence of a recursive set.

* Received by the editors September 6, 1988; accepted for publication (in revised form) September 18, 1989.

† Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218. This research was supported in part by the National Science Foundation under grants CCR-8808949 and CCR-8958528, and done in part while the author was at Stanford University, supported by a fellowship from the Fannie and John Hertz Foundation. Present address, Department of Computer Science, Yale University, P.O. Box 2158, Yale Station, New Haven, Connecticut 06520.

[1] If the binary expansion of x is not unique, then x is rational, and there are algorithms to compute both expansions of x, so we say that x is recursive.

We use the word monotone to mean nondecreasing.

THEOREM 3. *Let f be a monotone, recursive function such that* $\mathrm{Kr}(f)$ *is recursive. There is an algorithm that solves the unbounded searching problem by asking $f(n)$ questions if and only if* $\mathrm{Kr}(f) \leqq 1$.

Proof. Suppose that there is an unbounded searching algorithm that asks $f(n)$ questions. Bentley and Yao's result implies that $\mathrm{Kr}(f) \leqq 1$. Conversely, let

$$s(j) = \sum_{1 \leqq i \leqq j} 2^{-f(i)}.$$

Let

$$g(x) = \min \{j: s(j) > x\}.$$

We determine n by determining $s(n)$. A single question tells us whether $n < g(x)$, and hence whether $s(n) \leqq x$ (if $g(x)$ is undefined, then we know without asking any questions that $s(n) \leqq x$, so we ask an arbitrary (rhetorical) question). Thus we can determine $s(n)$ by performing a binary search on the half-open interval $(0, 1]$: First we ask if $n < g(\frac{1}{2})$; based on the answer to that question, we either ask if $n < g(\frac{1}{4})$ or if $n < g(\frac{3}{4})$, etc.

After asking $f(n)$ questions, the binary search determines that $s(n)$ belongs to an interval $(a, b]$ where $b - a = 2^{f(n)}$. Thus

$$0 = 2^{f(n)}b - 2^{f(n)}b \leqq 2^{f(n)}b - 2^{f(n)}s(n) < 2^{f(n)}(b-a) = 1.$$

Because

$$2^{f(n)}s(n) = \sum_{1 \leqq i \leqq n} 2^{f(n)-f(i)}$$

and f is monotone, $2^{f(n)}s(n)$ is an integer. Because b is a division point from the binary-search algorithm, $2^{f(n)}b$ is an integer. Since their difference belongs to $[0, 1)$, the integers $2^{f(n)}s(n)$ and $2^{f(n)}b$ must be equal. Therefore $s(n) = b$. Therefore $s(n+1)$ cannot belong to $(a, b]$. Thus $n = g(s(n)) - 1 = g(b) - 1$, so n is determined by asking exactly $f(n)$ questions. □

The existence of an unbounded searching algorithm that asks $f(n)$ questions is equivalent to the existence of an infinite recursive left-finite binary tree such that the nth leaf in the tree is at depth $f(n)$ for all n (see Definition 6 and Prop. 7). For finite trees, the preceding theorem is equivalent to Gilbert's theorem that the optimal alphabetic tree with weights in sorted order is an optimal Huffman tree.

In [Knu81], Knuth proved a result about unbounded searching schemes that ask no irredundant questions, i.e., no questions to which the answer could be deduced from previous answers.

THEOREM 4 (Knuth). *Let f be a monotone, recursive function. There is an algorithm that solves the unbounded searching problem by asking $f(n)$ irredundant questions if and only if* $\mathrm{Kr}(f) = 1$.

Since the number 1.0 is recursive, the existence of an unbounded searching algorithm follows from Theorem 3. However, Knuth's proof is especially beautiful because it produces a closed form for the sequence of questions asked by the unbounded searching algorithm.

Knuth proved the existence of extremely good unbounded searching algorithms by constructing series whose sum is exactly one (see [RS88] for some improvements by Reingold and Shen). We, on the other hand, need only to construct series whose sum is less than or equal to one, provided that we can show that the sum is recursive. Lemma 5 below shows that if we have an effective bound (approaching zero) on the

tail of a series, then the sum of that series is recursive. Because proofs of convergence usually provide such an effective bound, Lemma 5 implies that the effective-computability requirement can usually be verified without extra effort.

We write Q^+ to denote the set of positive rational numbers.

LEMMA 5. *Let g and h be total recursive functions from Z^+ to Q^+ such that*

$$\lim_{n\to\infty} h(n) = 0,$$

and

$$\sum_{i>n} g(i) < h(n).$$

Then $\sum_{i\geqq 1} g(i)$ is recursive.

Proof.

$$\lim_{n\to\infty} \sum_{i>n} g(i) \leqq \lim_{n\to\infty} h(n) = 0.$$

Therefore $\sum_{i\geqq 1} g(i)$ converges, so let

$$\sum_{i\geqq 1} g(i) = w.$$

We will show how to determine for any rational number y whether $w < y$. If w is rational, then we can simply compare the two rational numbers w and y. Otherwise, assume that w is irrational. Therefore $w \neq y$. Since $h(n)$ approaches 0, we can find the least n such that

$$\sum_{1\leqq i\leqq n} g(i) \geqq y \quad \text{or} \quad h(n) + \sum_{1\leqq i\leqq n} g(i) < y.$$

In the first case $w > y$. In the second case $w < y$. □

We end this section with a straightforward characterization of unbounded searching algorithms. This characterization will be useful in § 5.

DEFINITION 6. Let T be a recursive subset of $\{0, 1\}^*$. T is a *recursive binary tree* if, for all strings σ,

$$((\sigma 0 \in T) \quad \text{or} \quad (\sigma 1 \in T)) \Rightarrow (\sigma \in T).$$

T is a *recursive, left-finite binary tree* if, in addition, T has no infinite path except for its rightmost path.

PROPOSITION 7. *There is an unbounded searching algorithm that asks exactly $f(n)$ questions if and only if there is a recursive left-finite binary tree whose nth leaf from the left has depth $f(n)$ for all n.*

Proof. Suppose that there is an unbounded searching algorithm that asks exactly $f(n)$ questions. As noted in [BY76], it is easy to see that there exists a prefix-free encoding of the positive integers that uses $f(n)$ bits to encode n. Furthermore, that encoding is computable and lexicographic. We let the tree T consist of all strings that encode a positive integer, and all prefixes of such strings. It is easy to see that T has the desired properties.

Conversely, suppose that T is a recursive binary tree whose nth leaf from the left has depth $f(n)$ for all n. If the root of T has only one child, then the unbounded searching algorithm asks a rhetorical question. If the root has two children, then the left child has only finitely many descendants, so the algorithm asks if n is greater than the left child's rightmost descendant. The algorithm chooses the left subtree or the right subtree accordingly and continues recursively. □

2.1. Recursion theoretic aspects. One might ask whether all of the assumptions in Theorem 3 are necessary. In order to answer this question, it is helpful to know which

real numbers can be expressed in the form $\mathrm{Kr}(f)$ for some recursive function f. We identify each real number x in the interval $[0, 1)$ with the set of positive integers whose characteristic sequence is obtained by omitting the decimal point from the binary representation of x. Thus we can talk about a real number being recursive, recursively enumerable, etc.

In this section, we assume some familiarity with recursion theory. Let K denote the halting set. We write $A(z)=1$ if $z\in A$, and $A(z)=0$ if $z\notin A$. We write $x(z)$ to denote the zth bit in the binary expansion of the real number x.

PROPOSITION 8. (i) *If x is recursively enumerable, then there exists a monotone, recursive function f such that $x=\mathrm{Kr}(f)$.*

(ii) *If $\mathrm{Kr}(f)+\mathrm{Kr}(g)$ is recursive, then $\mathrm{Kr}(f)$ is recursive and $\mathrm{Kr}(g)$ is recursive.*

(iii) *If x is nonrecursive and corecursively enumerable, then x is not equal to $\mathrm{Kr}(f)$ for any recursive f.*

(iv) *If f is recursive, then $\mathrm{Kr}(f)$ is truth-table reducible to K.*

(v) *If x is truth-table reducible to K, then there is a monotone, recursive function f such that x is 1-reducible to $\mathrm{Kr}(f)$.*

Proof. (i) Let A be the recursively enumerable set corresponding to x. If A is finite, then f is easy to construct, so assume that A is infinite. Then there exists a total recursive 1-1 function g whose range is A. We define f by an easy initial segment argument. At stage s we will contribute 2^{-z} to $\mathrm{Kr}(f)$ where z is the sth element of A. Thus $\mathrm{Kr}(f)$ will be the real number corresponding to the characteristic sequence of A.

Stage 0. Let $n=1$. Let $k=1$.

Stage $s+1$. Let $z=g(s)$. Let $k=\max(k, z)$. Let $t=2^{k-z}$. For $n\leqq i<n+t$, define $f(i)=k$. Finally, let $n=n+t$. (This contributes $t2^{-k}=2^{-z}$ to $\mathrm{Kr}(f)$.)

Then, for $z\geqq 1$, $\mathrm{Kr}(f)(z)=A(z)$.

(ii) Let

$$t_k=\sum_{1\leqq n\leqq k}2^{-f(n)},$$

$$u_k=\sum_{1\leqq n\leqq k}2^{-g(n)}.$$

Then $\mathrm{Kr}(f)(z)=\lim_{k\to\infty}t_k(z)$, and $\mathrm{Kr}(g)(z)=\lim_{k\to\infty}u_k(z)$. We will compute $\mathrm{Kr}(f)(z)$ and $\mathrm{Kr}(g)(z)$ by choosing k large enough so that the first z bits of t_k and u_k have converged to their limiting value. Let $s_k=t_k+u_k$. Then $\{s_k\}_{k=1,2,\ldots}$ approaches $\mathrm{Kr}(f)+\mathrm{Kr}(g)$ from below. Find k such that the first z bits of the larger binary expansion of s_k agree with the first z bits of the smaller binary expansion of $\mathrm{Kr}(f)+\mathrm{Kr}(g)$. For any $j\geqq k$, the first z bits of t_j and u_j must agree with the first z bits of t_k and u_k, respectively, for otherwise $s_j>\mathrm{Kr}(f)+\mathrm{Kr}(g)$. Thus $\mathrm{Kr}(f)(z)=t_k(z)$ and $\mathrm{Kr}(g)(z)=u_k(z)$.

(iii) By part (i), there is a recursive function g such that $1-x=\mathrm{Kr}(g)$. If there is a recursive function f such that $x=\mathrm{Kr}(f)$, then $\mathrm{Kr}(f)+\mathrm{Kr}(g)=1$, which is recursive, so $\mathrm{Kr}(f)$ is recursive by part (ii).

(iv) Let $s_k=\sum_{1\leqq n\leqq k}2^{-f(n)}$. Let $g(z, k)=s_k(z)$. Then $\mathrm{Kr}(f)(z)=\lim_{k\to\infty}g(z, k)$. Since $g(z, k+1)\neq g(z, k)$ for at most 2^z-1 values of k, $\mathrm{Kr}(f)$ is truth-table reducible to K.

(v) Since x is truth-table reducible to K, there are recursive functions g and c such that $g(z, 0)=0$, $x(z)=\lim_{s\to\infty}g(z, s)$, and $|\{s: g(z, s)\neq g(z, s+1)\}|\leqq c(z)$. We define f by an easy initial segment argument.

Stage -1. Let $b_0=1$. Let $k=1$. Let $n=1$.

Stage $\langle z, 0\rangle$. Let $t = \lceil \log_2 c(z)+1 \rceil$. Let $b_{z+1} = b_z + t$.

Stage $\langle z, s+1\rangle$. If $g(z, s+1) = g(z, s)$, then go to the next stage. Otherwise, let $w = b_{z+1} - 1$. Let $k = \max(k, w)$. Let $n' = n + 2^{k-w}$. Let $f(m) = k$ for $n \leqq m < n'$. Let $n = n'$.

At stage $\langle z, 0\rangle$ we decide to store $x(z)$ in $\mathrm{Kr}(f)(w)$, where $w = b_{z+1} - 1$. Whenever $g(z, s+1) \neq g(z, s)$, we contribute $2^{k-w}2^{-k} = 2^{-w}$ to the value of $\mathrm{Kr}(f)$, so as to complement $\mathrm{Kr}(f)(w)$. But at stage $\langle z+1, 0\rangle$ we carefully set aside a long enough block of unused bits so that there will be no overflow from the location storing the $x(z+1)$ into the location storing $x(z)$. Thus $x(z) = \mathrm{Kr}(f)(b_{z+1} - 1)$. □

Although we know a lot about which real numbers are $\mathrm{Kr}(f)$ for some f, it would be interesting to characterize that class of real numbers exactly.

COROLLARY 9. *There exists a monotone, recursive function f such that* $\mathrm{Kr}(f) < 1$, *but* $\mathrm{Kr}(f)$ *is not recursive.*

Proof. Let x be a nonrecursive recursively enumerable real number in part (i) of the proposition above. □

PROPOSITION 10. *If there is an algorithm that solves the unbounded searching problem by asking exactly $f(n)$ questions, then*

(i) *f is recursive.*

(ii) $\mathrm{Kr}(f)$ *is recursive.*

Proof. (i) To compute $f(n)$, run the unbounded searching algorithm on n. $f(n)$ is the number of questions asked.

(ii) $\mathrm{Kr}(f)$ is approximated from below by the sequence $\{s_k\}_{k\geqq 1}$ where $s_k = \sum_{1\leqq n\leqq k} 2^{-f(n)}$.

We can also approximate $\mathrm{Kr}(f)$ from above. For $k \geqq 1$, let n_k be the number obtained by the unbounded searching algorithm when the first k answers are "yes" and all remaining answers are "no." Then $\sum_{n\geqq n_k} 2^{-f(n)} < 2^{-k}$. Let $t_k = \sum_{1\leqq n<n_k} 2^{-f(n)} + 2^{-k}$. Then $\mathrm{Kr}(f)$ is approximated from above by $\{t_k\}_{k\geqq 1}$. To compute $\mathrm{Kr}(f)(z)$ find the least k such that the first k bits of the larger binary expansion of s_k agree with the first k bits of the smaller binary expansion of t_k. Then $\mathrm{Kr}(f)(z) = s_k(z)$. □

2.2. Are all conditions necessary? We show that the requirement that f be monotone in Theorem 3 is not necessary, but not superfluous.

PROPOSITION 11. (i) *There is a nonmonotone function f such that there is an unbounded searching algorithm that asks exactly $f(n)$ questions.*

(ii) *There is a nonmonotone, recursive function f such that* $\mathrm{Kr}(f) = 1$, *but there is no unbounded searching algorithm that asks at most $f(n)$ questions.*

Proof. (i) Let

$$f(n) = \begin{cases} 3 & \text{if } n = 1 \quad \text{or} \quad n = 2, \\ 2 & \text{if } n = 3, \\ n-2 & \text{otherwise.} \end{cases}$$

It is readily verified that there is an unbounded searching algorithm that asks exactly $f(n)$ questions.

(ii) Let

$$f(n) = \begin{cases} 2 & \text{if } n = 1, \\ 1 & \text{if } n = 2, \\ n & \text{otherwise.} \end{cases}$$

It is readily verified that there is no unbounded searching algorithm that asks at most $f(n)$ questions, and that $\mathrm{Kr}(f)=1$. □

A corollary to Proposition 10 is the following strengthened version of Theorem 3.

THEOREM 3′. *Let f be a monotone function. There is an algorithm that solves the unbounded searching problem by asking exactly $f(n)$ questions if and only if f is recursive,* $\mathrm{Kr}(f)\leqq 1$, *and* $\mathrm{Kr}(f)$ *is recursive.*

An important question is whether the assumptions are still necessary if we require merely that the unbounded searching algorithm ask $\leqq f(n)$ questions, rather than exactly $f(n)$ questions. We show that in this case, the assumption that f is recursive is not necessary, but not superfluous.

PROPOSITION 12. (i). *There exists a nonrecursive, monotone function f with* $\mathrm{Kr}(f)<1$ *such that there exists an unbounded searching algorithm that asks fewer than $f(n)$ questions.*

(ii) *There exists a nonrecursive, monotone function f with* $\mathrm{Kr}(f)=1$ *such that there exists no unbounded searching algorithm that asks at most $f(n)$ questions.*

Proof. (i) Let A be any nonrecursive set. Let g be any recursive function such that there is an unbounded searching algorithm that asks exactly $g(n)$ questions. Let $f(n)=g(n)+p_A(n)$, where $p_A(n)$ is the nth smallest element of A. Then $A\leqq_T f$ so f is nonrecursive, $g(n)<f(n)$ for all n, and f is monotone because g and p_A are monotone.

(ii) Let A be any nonrecursive set. We will construct f such that $A\leqq_T f$. Define f as follows: If $n\in A$, then $f(12n-11)=n+2$, $f(12n-10)=n+3$, and $f(12n-9)=f(12n-8)=n+4$. If $n\notin A$ then $f(12n-11)=f(12n-10)=f(12n-9)=f(12n-8)=n+3$. In either case, $f(12n-7)=\cdots=f(12n)=n+4$. Since $\mathrm{Kr}(f)=1$, there is no unbounded searching algorithm that asks fewer than $f(n)$ questions. Since f is nonrecursive, there is no unbounded searching algorithm that asks exactly $f(n)$ questions. □

Open Question 13. *Let f be a recursive, monotone function such that $Kr(f)$ is nonrecursive and less than one. When is there an unbounded searching algorithm that asks fewer than $f(n)$ questions?*

This question could be restated as follows: When does there exist a recursive, monotone function g such that f dominates g and $\mathrm{Kr}(g)$ is recursive? Functions satisfying the stated assumptions exist by Proposition 8(v); however, the answer is not known for any such function. If the answer is "yes" for all f, then we could simplify Theorem 3 by omitting the requirement that $\mathrm{Kr}(f)$ be recursive, and instead allow the unbounded searching algorithm to ask $\leqq f(n)$ questions. If the answer is "no" for all f, then Theorem 3 is tight even if we allow the unbounded searching algorithm to ask $\leqq f(n)$ questions.

3. Some slowly converging series. In this section we show that the upper bounds obtained by Bentley and Yao follow from Theorem 3. We also obtain improved upper bounds. Good upper bounds depend on series that converge slowly; we can define such series in terms of iterated logarithms.

DEFINITION 14.

$$\log_b^{(i)}(x)=\begin{cases}x & \text{if } i=0,\\ \log_b \log_b^{(i-1)}(x) & \text{otherwise.}\end{cases}$$

DEFINITION 15.

$$\log_b^{*} x=\min\{t\colon \log_b^{(t)}(x)\leqq 1\}.$$

DEFINITION 16.

$$\text{logsum}_b(x) = \sum_{1 \leqq i \leqq \log_b^* x} \log_b^{(i)}(x).$$

DEFINITION 17.

$$\text{logprod}_b(x) = \prod_{0 \leqq i < \log_b^* x} \log_b^{(i)}(x).$$

Note that $\text{logsum}_b(x) = \log_b(\text{logprod}_b(x))$.

We will use the following lemma when approximating a sum by an integral.

LEMMA 18.

$$\frac{1}{\prod_{0 \leqq i \leqq n} \log_b^{(i)}(x)} = \frac{d}{dx}((\ln b)^{n+1} \log_b^{(n+1)}(x)).$$

Proof. This follows from the chain rule for differentiation. □

DEFINITION 19.

$$\text{tow}_b(n) = \begin{cases} 1 & \text{if } n = 0, \\ b^{\text{tow}_b(n-1)} & \text{otherwise.} \end{cases}$$

The next lemma will allow us to show that $\sum 1/(\text{logprod}_2(k) c^{\log_2^* k})$ converges when $c > \ln 2$.

LEMMA 20. *For every function g,*

$$\sum_{\text{tow}_2(n) < k \leqq \text{tow}_2(n+1)} \frac{1}{\text{logprod}_2(k) g(\log_2^* k)} \leqq \frac{(\ln 2)^{n+1}}{g(n+1)}.$$

Proof.

$$\sum_{\text{tow}_2(n) < k \leqq \text{tow}_2(n+1)} \frac{1}{\text{logprod}_2(k) g(\log_2^* k)}$$

$$= \sum_{\text{tow}_2(n) < k \leqq \text{tow}_2(n+1)} \frac{1}{(\prod_{0 \leqq j < \log_2^* k} \log_2^{(j)}(k)) g(\log_2^* k)}$$

by the definition of logprod

$$= \sum_{\text{tow}_2(n) < k \leqq \text{tow}_2(n+1)} \frac{1}{(\prod_{0 \leqq j < n+1} \log_2^{(j)}(k)) g(n+1)}$$

because $\log^* k = n+1$

$$\leqq \frac{1}{g(n+1)} \int_{\text{tow}_2(n)}^{\text{tow}_2(n+1)} \left(\frac{1}{\prod_{0 \leqq j < n+1} \log_2^{(j)} x} \right) dx$$

by the rectangle rule for sums

$$= \frac{1}{g(n+1)} (\ln 2)^{n+1} \log_2^{(n+1)}(x) \Big|_{\text{tow}_2(n)}^{\text{tow}_2(n+1)} \quad \text{by Lemma 18}$$

$$= \frac{1}{g(n+1)} (\ln 2)^{n+1} - 0$$

$$= \frac{(\ln 2)^{n+1}}{g(n+1)}.$$

□

LEMMA 21. (i) *Let* $c > \ln 2$.

$$\sum_{k > \text{tow}_2(t)} \frac{1}{\text{logprod}_2(k) c^{\log_2^* k}} \leqq \left(\frac{\ln 2}{c} \right)^{t+1} \left(\frac{1}{1 - (\ln 2/c)} \right).$$

(ii)

$$\sum_{k\geqq 1}\frac{1}{\operatorname{logprod}_2(k)}\leqq\frac{1}{1-\ln 2}.$$

Proof. (i)

$$\begin{aligned}\sum_{k>\operatorname{tow}_2(t)}\frac{1}{\operatorname{logprod}_2(k)c^{\log_2^* k}}&=\sum_{n\geqq t}\sum_{\operatorname{tow}_2(n)<k\leqq\operatorname{tow}_2(n+1)}\frac{1}{\operatorname{logprod}_2(k)c^{\log_2^* k}}\\&\leqq\sum_{n\geqq t}\left(\frac{\ln 2}{c}\right)^{n+1}\quad\text{by Lemma 20}\\&=\left(\frac{\ln 2}{c}\right)^{t+1}\left(\frac{1}{1-(\ln 2/c)}\right).\end{aligned}$$

(ii)

$$\begin{aligned}\sum_{k\geqq 1}\frac{1}{\operatorname{logprod}_2(k)}&=\frac{1}{\operatorname{logprod}_2(1)}+\sum_{k>1}\frac{1}{\operatorname{logprod}_2(k)}\\&=1+\sum_{k>\operatorname{tow}(0)}\frac{1}{\operatorname{logprod}_2(k)}\\&\leqq 1+\frac{\ln 2}{1-\ln 2}\quad\text{by part (i)}\\&=\frac{1}{1-\ln 2}.\end{aligned}$$

□

THEOREM 22. (i) *There is an unbounded searching algorithm that asks exactly* $\lceil\operatorname{logsum}_2(n)+2\rceil$ *questions.*

(ii) *Let e be the base of the natural logarithms, let* $0<\varepsilon<e$, *and let*

$$f(n)=\lceil\operatorname{logsum}_2(n)-(\log_2\log_2(e-\varepsilon))\log_2^* n\rceil.$$

There is an unbounded searching algorithm that asks $f(n)+O(1)$ *questions.*

Proof. (i) Because $f(n)\geqq\operatorname{logsum}_2(n)+2$,

$$\begin{aligned}\operatorname{Kr}(f)&\leqq\sum_{n\geqq 1}\frac{1}{4\operatorname{logprod}_2(n)}\\&\leqq\frac{1}{4(1-\ln 2)}\quad\text{by Lemma 21(ii)}\\&<1\quad\text{by direct calculation.}\end{aligned}$$

Lemma 21(i) provides an effective bound on the tail of the sum. Thus Lemma 5 and Theorem 3 imply the existence of an unbounded searching algorithm that asks $f(n)$ questions.

(ii) Because $f(n)\geqq\operatorname{logsum}_2(n)-(\log_2\log_2(e-\varepsilon))\log_2^* n$,

$$\begin{aligned}\sum_{n>\operatorname{tow}_2(t)}2^{-f(n)}&\leqq\sum_{n>\operatorname{tow}_2(t)}\frac{(\log_2(e-\varepsilon))^{\log_2^* n}}{\operatorname{logprod}_2(n)}\\&\leqq\left(\frac{\ln 2}{\log_{e-\varepsilon}2}\right)^{t+1}\left(\frac{1}{1-(\ln 2/\log_{e-\varepsilon}2)}\right)\quad\text{by Lemma 21(i)}\\&=(\ln(e-\varepsilon))^{t+1}\left(\frac{1}{1-\ln(e-\varepsilon)}\right).\end{aligned}$$

This provides an effective bound on the tail of the series, so $\mathrm{Kr}(f)$ converges. Let its sum be at most 2^s, where s is a positive integer. Then

$$\sum_{n \geqq 1} 2^{-(f(n)+s)} \leqq 1$$

and we have an effective bound on the tail of the sum. By Lemma 5 and Theorem 3, there is an unbounded searching algorithm that asks at most $f(n)+s$ questions. □

These searching algorithms are comparable to Bentley and Yao's "ultimate" algorithm U [BY76]. Bentley and Yao's algorithm is more practical than ours because theirs was produced constructively. However, it is interesting that we can derive ours directly from Theorem 3 and lemmas about convergence. Our algorithms are also interesting because there is a remark in [BY76], attributed to Chung and Graham, stating that unbounded search could not be accomplished with as few as

$$\mathrm{logsum}_2 n + \log_2 \log_2 (\log^* n) + O(1)$$

questions; we, however, have shown that unbounded search can be accomplished with asymptotically fewer than $\mathrm{logsum}_2 n$ questions. (A related comment appears in [RS88].)

3.1. Other bases. We can prove similar results using bases different from 2.

LEMMA 23. *For all* $b \geqq 2$,

$$\sum_{\mathrm{tow}_b(n) < k \leqq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \leqq b^{-n} + (\ln b)^{n+1}.$$

Proof.

$$\begin{aligned}
&\sum_{\mathrm{tow}_b(n) < k \leqq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \\
&= \sum_{\lfloor \mathrm{tow}_b(n) \rfloor < k \leqq \lfloor \mathrm{tow}_b(n+1) \rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
&= \frac{1}{\mathrm{logprod}_b(\lfloor \mathrm{tow}_b(n) \rfloor + 1)} + \sum_{\lfloor \mathrm{tow}_b(n) \rfloor + 1 < k \leqq \lfloor \mathrm{tow}_b(n+1) \rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
&\leqq \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \sum_{\lfloor \mathrm{tow}_b(n) \rfloor + 1 < k \leqq \lfloor \mathrm{tow}_b(n+1) \rfloor} \frac{1}{\mathrm{logprod}_b(k)} \\
&\leqq \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \int_{\lfloor \mathrm{tow}_b(n) \rfloor + 1}^{\lfloor \mathrm{tow}_b(n+1) \rfloor} \left(\frac{1}{\mathrm{logprod}_b(x)} \right) dx \\
&\leqq \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \int_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \left(\frac{1}{\mathrm{logprod}_b(x)} \right) dx \\
&= \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + \left((\ln b)^{n+1} \log_b^{(n+1)}(x) \Big|_{\mathrm{tow}_b(n)}^{\mathrm{tow}_b(n+1)} \right) \quad \text{by Lemma 18} \\
&= \frac{1}{\mathrm{logprod}_b(\mathrm{tow}_b(n))} + (\ln b)^{n+1} \\
&\leqq \frac{1}{\mathrm{tow}_b(n)} + (\ln b)^{n+1} \\
&\leqq \frac{1}{b^n} + (\ln b)^{n+1}.
\end{aligned}$$

□

COROLLARY 24. *If* $2 \leqq b < e$, *then*

(i)

$$\sum_{k>\mathrm{tow}_b(t)} \frac{1}{\mathrm{logprod}_b(k)} \leqq \frac{b^{-t}}{1-(1/b)} + \frac{(\ln b)^{t+1}}{1-\ln b}.$$

(ii)

$$\sum_{k\geqq 1} \frac{1}{\mathrm{logprod}_b(k)} \leqq \frac{b}{b-1} + \frac{1}{1-\ln b}.$$

Proof.

(i)

$$\begin{aligned}\sum_{k>\mathrm{tow}_b(t)} \frac{1}{\mathrm{logprod}_b(k)} &= \sum_{n\geqq t} \sum_{\mathrm{tow}_b(n)<k\leqq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \\ &\leqq \sum_{n\geqq t} (b^{-n} + (\ln b)^{n+1}) \quad \text{by Lemma 23} \\ &= \frac{b^{-t}}{1-(1/b)} + \frac{(\ln b)^{t+1}}{1-\ln b}.\end{aligned}$$

(ii)

$$\begin{aligned}\sum_{k\geqq 1} \frac{1}{\mathrm{logprod}_b(k)} &= \frac{1}{\mathrm{logprod}_b(1)} + \sum_{k>1} \frac{1}{\mathrm{logprod}_b(k)} \\ &= 1 + \sum_{k>\mathrm{tow}(0)} \frac{1}{\mathrm{logprod}_b(k)} \\ &\leqq 1 + \frac{1}{1-(1/b)} + \frac{\ln b}{1-\ln b} \quad \text{by part (i)} \\ &= \frac{b}{b-1} + \frac{1}{1-\ln b} \quad \text{by arithmetic.}\end{aligned}$$

□

THEOREM 25. *If* $2 < b < e$, *then there is an unbounded searching algorithm that asks* $f(n) = \lceil (\log_2 b)\,\mathrm{logsum}_b(n) + \log_2((b/b-1)+(1/1-\ln b))\rceil$ *questions.*

Proof.

$$\begin{aligned}\mathrm{Kr}(f) &\leqq \sum_{n\geqq 1} \frac{1}{\mathrm{logprod}_b(n)} \frac{1}{(b/b-1)+(1/1-\ln b)} \\ &\leqq 1 \text{ by Corollary 24(ii).}\end{aligned}$$

Corollary 24(ii) provides an effective bound on the tail of the sum. Thus Lemma 5 and Theorem 3 imply the existence of an unbounded searching algorithm that asks $f(n)$ questions. □

4. Some slowly diverging series. In this section we produce some series that diverge very slowly, in order to prove some strong lower bounds on unbounded search.

LEMMA 26. *For all* $b \geqq 2$,

$$\sum_{\mathrm{tow}_b(n)<k\leqq \mathrm{tow}_b(n+1)} \frac{1}{\mathrm{logprod}_b(k)} \geqq \frac{-1}{b^n-1} + (\ln b)^{n+1}.$$

Proof.

$$\sum_{\text{tow}_b(n)<k\leqq\text{tow}_b(n+1)} \frac{1}{\text{logprod}_b(k)}$$

$$= \sum_{\lfloor\text{tow}_b(n)\rfloor<k\leqq\lfloor\text{tow}_b(n+1)\rfloor} \frac{1}{\text{logprod}_b(k)}$$

$$= \frac{-1}{\text{logprod}_b(\lfloor\text{tow}_b(n)\rfloor)} + \sum_{\lfloor\text{tow}_b(n)\rfloor\leqq k<\lfloor\text{tow}_b(n+1)\rfloor+1} \frac{1}{\text{logprod}_b(k)}$$

$$\geqq \frac{-1}{\text{logprod}_b(\lfloor\text{tow}_b(n)\rfloor)} + \int_{\lfloor\text{tow}_b(n)\rfloor}^{\lfloor\text{tow}_b(n+1)\rfloor+1} \left(\frac{1}{\text{logprod}_b(x)}\right) dx$$

$$\geqq \frac{-1}{\text{logprod}_b(\lfloor\text{tow}_b(n)\rfloor)} + \int_{\text{tow}_b(n)}^{\text{tow}_b(n+1)} \left(\frac{1}{\text{logprod}_b(x)}\right) dx$$

$$\geqq \frac{-1}{\lfloor\text{tow}_b(n)\rfloor} + \int_{\text{tow}_b(n)}^{\text{tow}_b(n+1)} \left(\frac{1}{\text{logprod}_b(x)}\right) dx$$

$$\geqq \frac{-1}{\text{tow}_b(n)-1} + \int_{\text{tow}_b(n)}^{\text{tow}_b(n+1)} \left(\frac{1}{\text{logprod}_b(x)}\right) dx$$

$$\geqq \frac{-1}{b^n-1} + \int_{\text{tow}_b(n)}^{\text{tow}_b(n+1)} \left(\frac{1}{\text{logprod}_b(x)}\right) dx$$

$$= \frac{-1}{b^n-1} + ((\ln b)^{n+1} \log_b^{(n+1)}(x)|_{\text{tow}_b(n)}^{\text{tow}_b(n+1)}) \quad \text{by Lemma 18}$$

$$= \frac{-1}{b^n-1} + (\ln b)^{n+1}. \qquad \square$$

LEMMA 27. *If $b \geqq e$, then*

$$\sum_{k\geqq 1} \frac{1}{\text{logprod}_b(k)}$$

diverges.

Proof.

$$\sum_{k\geqq 1} \frac{1}{\text{logprod}_b(k)} \geqq \sum_{n\geqq 0} \sum_{\text{tow}_b(n)<k\leqq\text{tow}_b(n+1)} \frac{1}{\text{logprod}_b(k)}$$

$$\geqq \sum_{n\geqq 0} \left(\frac{-1}{b^n-1} + (\ln b)^{n+1}\right) \quad \text{by Lemma 26.}$$

That sum diverges to positive infinity because its nth term approaches 1 (for $b = e$) or positive infinity (for $b > e$) in the limit. $\square$

THEOREM 28. *If $b \geqq e$, then there is no unbounded searching algorithm that asks at most*

$$f(n) = (\log_2 b)\,\text{logsum}_b\, n + c$$

questions, for any constant c.

Proof. Assume the contrary. Then Theorem 1 implies that $\text{Kr}(f) \leqq 1$. However, Lemma 27 implies that $\text{Kr}(f)$ diverges. $\square$

The preceding result shows that Theorem 25 is rather tight.

LEMMA 29. *If*

$$\sum_{k\geqq 1} \frac{1}{h(k)}$$

diverges, then

$$\sum_{k\geqq 1} \frac{1}{\operatorname{logprod}_2(k)(\ln(2))^{\log_2^*(k)} h(\log_2^*(k))}$$

diverges.

Proof.

$$\begin{aligned}
&\sum_{k\geqq 1} \frac{1}{\operatorname{logprod}_2(k)(\ln 2)^{\log_2^*(k)} h(\log_2^*(k))} \\
&\quad = \sum_{n\geqq 0} \sum_{\operatorname{tow}_2(n)<k\leqq \operatorname{tow}_2(n+1)} \frac{1}{\operatorname{logprod}_2(k)(\ln 2)^{\log_2^*(k)} h(\log_2^*(k))} \\
&\quad = \sum_{n\geqq 0} \sum_{\operatorname{tow}_2(n)<k\leqq \operatorname{tow}_2(n+1)} \frac{1}{\operatorname{logprod}_2(k)(\ln 2)^{n+1} h(n+1)} \\
&\quad = \sum_{n\geqq 0} \frac{1}{(\ln 2)^{n+1} h(n+1)} \sum_{\operatorname{tow}_2(n)<k\leqq \operatorname{tow}_2(n+1)} \frac{1}{\operatorname{logprod}_2(k)} \\
&\quad \geqq \sum_{n\geqq 0} \frac{1}{(\ln 2)^{n+1} h(n+1)} \left(\frac{-1}{2^n-1} + (\ln 2)^{n+1}\right) \quad \text{by Lemma 26.} \\
&\quad = \sum_{n\geqq 0} \frac{1}{h(n+1)} \left(\frac{-(\log_2 e)^{n+1}}{2^n-1} + 1\right).
\end{aligned}$$

For large n the nth term of the sum approaches $1/(h(n+1))$. Therefore the sum diverges. □

COROLLARY 30.

$$\sum_{k\geqq 1} \frac{1}{\operatorname{logprod}_2(k)(\ln(2))^{\log_2^*(k)}}$$

diverges.

Proof. Let $h(n)=1$ in Lemma 29. □

The following theorem improves a lower bound from [BY76].

THEOREM 31. *There is no unbounded searching algorithm that asks at most $f(n)$ questions, where*

$$f(n) = \operatorname{logsum}_2 n - (\log_2 \log_2 e)\log_2^* n + c,$$

for any constant c.

Proof. Assume the contrary. Then Theorem 1 implies that $\operatorname{Kr}(f) \leqq 1$. However, Lemma 26 implies that $\operatorname{Kr}(f)$ diverges. □

The lower bound from the preceding theorem and the upper bound from Theorem 22(ii) differ by less than any positive constant times $\log_2^* n$. In [Knu81], Knuth has obtained bounds that differ by less than $(\log_2^*)^{(m)} n$ for any m; therefore, at least one of Knuth's bounds is superior to ours. His bounds involve iterates of the function $\lfloor \log_2 n \rfloor$; it would be interesting to know if both of Knuth's bounds are superior to ours. In the next section we will see how to produce algorithms that are as close as practically possible to being optimal.

5. Nearly optimal algorithms. The algorithms presented by ourselves and by others are very close to optimal. In this section, we will show that given any algorithm for unbounded search, there is an asymptotically better algorithm. In contrast, a result of

Raoult and Vuillemin [RV79] states that there exist algorithms that are as close as we like to optimal.

There is an unbounded searching algorithm with $f(n) = n$. Because

$$\sum_{n \geqq 1} 2^{-n} = 1,$$

that algorithm cannot be improved for any value of n without making it worse for some other value of n. However, we have already seen algorithms that are asymptotically much better. If we are willing to ask a larger number of questions to determine the first few numbers, then we can improve that algorithm asymptotically by an additive constant c. Furthermore, given any unbounded searching algorithm that asks exactly $f(n)$ questions, where f is monotone, we can easily improve that algorithm asymptotically by an additive constant c. Choose t such that

$$\sum_{n > t} 2^{-f(n)} \leqq 2^{-(c+1)},$$

choose m such that $t2^{-m} \leqq \frac{1}{2}$, and let

$$g(n) = \begin{cases} m & \text{if } 1 \leqq n \leqq t, \\ \max(f(n) - c, m) & \text{if } n > t. \end{cases}$$

The function g is monotone, because f is monotone. Furthermore,

$$\begin{aligned} \mathrm{Kr}(g) &= \sum_{1 \leqq n \leqq t} 2^{-g(n)} + \sum_{n > t} 2^{-g(n)} \\ &\leqq \tfrac{1}{2} + \sum_{n > t} 2^{-g(n)} \\ &\leqq \tfrac{1}{2} + \sum_{n > t} 2^{-(f(n)-c)} \\ &\leqq \tfrac{1}{2} + 2^{c} 2^{-(c+1)} \\ &= 1. \end{aligned}$$

We note that $\mathrm{Kr}(g)$ is recursive because $\mathrm{Kr}(f)$ is recursive and $f(n) - g(n)$ is constant for large n. An effective bound on the tail of the sum is easily found. Therefore we can perform unbounded search by asking exactly $g(n)$ questions.

In fact, we can improve every unbounded searching algorithm asymptotically by more than a constant.

THEOREM 32. *Suppose that we can perform unbounded search by asking exactly $f(n)$ questions. Then there exists a function g such that we can perform unbounded search by asking $g(n)$ questions and such that*

$$\lim_{n \to \infty} f(n) - g(n) = \infty.$$

If f were monotone, then we could prove this result in the same way as the preceding result. However, we do not assume that f is monotone. We prove the result via tree rotations.

Proof. By Proposition 7, let T be a recursive left-finite binary tree whose nth leaf has depth $f(n)$. Let v be the first node on T's infinite path that has exactly four grandchildren. Let a be the left child of v, and let b and c be the right grandchildren of v. We rotate the subtree rooted at v as follows: a and b become the left grandchildren

of v, and c becomes the right child of v. This has the effect of lowering finitely many leaves, but raising infinitely many leaves. We repeat the process recursively on c. Let $g(n)$ be the depth of the nth leaf in the recursive tree produced in this way. It is easy to verify that g has the desired properties. □

The next result shows that there exist unbounded searching algorithms that differ from the elusive optimum by less than any given recursive function g, provided that g grows towards infinity. Although it was originally proved by Raoult and Vuillemin, we include the proof for the sake of completeness.

THEOREM 33 [RV79]. *Let $g(n)$ be any monotone, recursive function such that*

$$\lim_{n\to\infty} g(n) = \infty.$$

Then there is a function f such that there is an unbounded searching algorithm that asks exactly $f(n)$ questions, but there is no unbounded searching algorithm that asks at most $f(n)-g(n)+c$ questions, for any constant c.

Proof. We construct $h = f - g$ via an easy initial segment argument.

Stage 0. Let $t_1 = 1$.

Stage $s \geqq 1$. Choose $k_s > k_{s-1}$ so that $g(t_s + 2^{k_s}) \geqq s+1$. Let $t_{s+1} = t_s + 2^{k_s}$. Let $h(n) = k_s + s - g(n)$ for $t_s \leqq n < t_{s+1}$.

By construction, $h(n)+g(n) = k_s + s$ for $t_s \leqq n < t_{s+1}$, and $k_{s+1} > k_s$. Therefore $h+g$ is monotone. In addition, since $t_{s+1} - t_s = 2^{k_s}$,

$$\sum_{t_s \leqq n < t_{s+1}} 2^{-(h(n)+g(n))} = 2^{k_s} 2^{-(k_s+s)} = 2^{-s}.$$

Summing over s, we find that $\mathrm{Kr}(h+g) = 1$. Consequently, there is an unbounded searching algorithm that asks exactly $h(n)+g(n)$ questions.

Also by construction, $h(n) \leqq k_s$ for $t_s \leqq n < t_{s+1}$. Therefore,

$$\sum_{t_s \leqq n < t_{s+1}} 2^{-h(n)} \geqq 1,$$

so $\mathrm{Kr}(h) = \infty$. Consequently, there is no unbounded searching algorithm that asks at most $h(n)+c$ questions for any constant c. □

6. Parallel unbounded search. Bentley and Yao suggest a variant of unbounded search where we are allowed to ask p simultaneous questions, i.e., in one round we are allowed to ask "Is $n \geqq t_1$?", "Is $n \geqq t_2$?", $\cdots$, "Is $n \geqq t_p$?". We call such questions higher-lower questions. How many rounds of higher-lower questions are needed to determine n? How many rounds of arbitrary questions are needed to determine n? The following solutions were suggested by Gasarch and Kruskal.

Theorem 1 generalizes easily to provide a lower bound.

THEOREM 34. *If $f(n)$ rounds of p questions suffice to solve the unbounded search problem, then*

$$\sum_{n \geqq 1} (p+1)^{-f(n)} \leqq 1.$$

Proof. As Bentley and Yao pointed out, there are only $p+1$ different possibilities for the result of a round of p questions of the form "Is $n \geqq t$?". Thus the results of all $f(n)$ rounds of questions form a prefix code for n in base p (see Bentley and Yao's proof when $p = 2$). The conclusion follows from Kraft's theorem [Gal68]. □

Theorem 3 generalizes easily to prove a corresponding upper bound.

THEOREM 35. *Let f be a monotone, recursive function such that*

$$\sum_{n \geq 1} (p+1)^{-f(n)}$$

is recursive and is less than or equal to one. Then there is an algorithm that solves the unbounded searching problem by asking $f(n)$ rounds of p questions.

Proof. Same as the proof of Theorem 3, except that we use $(p+1)$-ary search [Kru83] instead of binary search. □

The two theorems above allow us to generalize our earlier results by simply replacing $f(n)$ by $f(n)/\log_2(p+1)$.

- $\lceil (\log_{p+1} 2)(\text{logsum}_2(n) - (\log_2 \log_2 (e-\varepsilon)) \log_2^* n) \rceil$ rounds of p simultaneous higher-lower questions are sufficient for unbounded search.
- $(\log_{p+1} 2)(\text{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n) + O(1)$ rounds of p simultaneous higher-lower questions are not sufficient for unbounded search.
- Given any computable function g that grows without bound, there exists an unbounded searching algorithm (asking rounds of p simultaneous higher-lower questions) that is within $g(n)$ rounds of being optimal.

If we are allowed to ask rounds of p simultaneous questions, with no restriction on the form of the individual questions, then we may guess the number n after asking significantly fewer questions. In fact, if there is a recursive binary prefix code of length $f(n)$ for the positive integers, then we can guess n after asking $\lceil f(n)/p \rceil$ questions: Our ith question is "What are bits $(i-1)p+1$ through ip of the binary prefix code for n?" This bound is tight because the sequence of answers to all rounds of questions forms a recursive binary prefix code for n. This yields the following results:

- $\lceil (1/p)(\text{logsum}_2(n) - (\log_2 \log_2 (e-\varepsilon)) \log_2^* n) \rceil$ rounds of p simultaneous questions are sufficient for unbounded search.
- $(1/p)(\text{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n) + O(1)$ rounds of p simultaneous questions are not sufficient for unbounded search.
- Given any computable function g that grows without bound, there exists an unbounded searching algorithm (asking rounds of p simultaneous questions) that is within $g(n)$ rounds of being optimal.

7. Conclusions. If we are trying to guess a positive integer, then we can determine that the number is n by asking

$$f(n) = \lceil \text{logsum}_2(n) - (\log_2 \log_2 (e-\varepsilon)) \log_2^* n \rceil$$

questions of the form "Is n at least t?". However we cannot determine that the number is n by asking only

$$\text{logsum}_2 n - (\log_2 \log_2 e) \log_2^* n + O(1)$$

questions of that form. Similarly tight bounds can be obtained for parallel unbounded search.

Acknowledgments. This work was motivated by a question posed by Bill Gasarch in [Gas85] and solved in [BG]. Some of the ideas used in this paper are due to Bob Floyd's course on complexity theory, Stanford, 1983. I would also like to thank Bill Gasarch and Clyde Kruskal for helpful comments, especially on parallel unbounded search; Rao Kosaraju for pointing out the relationship to alphabetic coding; and Dan Willard for referring me to Bentley and Yao's paper.

REFERENCES

[BG] R. BEIGEL AND W. I. GASARCH, *On the complexity of finding the chromatic number of a recursive graph* II: *The unbounded case*, Ann. Pure Appl. Logic, to appear.

[BY76] J. L. BENTLEY AND A. C.-C. YAO, *An almost optimal algorithm for unbounded searching*, Inform. Process. Lett., 5 (1976), pp. 82–87.

[Gal68] P. E. GALLAGER, *Information Theory and Reliable Communication*, John Wiley, New York, 1968.

[Gas85] W. I. GASARCH, *A hierarchy of functions with applications to recursive graph theory*, Tech. Report TR-1651, Dept. of Computer Science, University of Maryland, College Park, MD, 1985.

[Knu81] D. E. KNUTH, *Supernatural numbers*, in The Mathematical Gardner, D. A. Klarner, ed., Wadsworth International, Belmont, CA, 1981, pp. 310–325.

[Kru83] C. P. KRUSKAL, *Searching, merging, and sorting in parallel computation*, IEEE Trans. Comput., C-32 (1983), pp. 942–946.

[RS88] E. M. REINGOLD AND X.-J. SHEN, *More nearly optimal algorithms for unbounded searching*, Tech. Report UIUCDCS-R-88-1471, Dept. of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL, November 1988.

[RV79] J.-C. RAOULT AND J. VUILLEMIN, *Optimal unbounded search strategies*, Tech. Report 33, Laboratoire de Recherche en Informatique, Université de Paris-Sud, Orsay, France, 1979.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 538–543, June 1990

011

THE EUCLIDEAN ALGORITHM AND THE DEGREE OF THE GAUSS MAP*

TAKIS SAKKALIS†

Abstract. This paper examines the computation of the topological degree of the Gauss map, defined by polynomials in the plane, via the Euclidean algorithm.

Key words. integer polynomials, topological degree, Gauss map

AMS(MOS) subject classifications. 68Q25, 55M20

1. Introduction. Let $p(x,y), q(x,y)$ be integer polynomials without common factors, of degrees n_1 and n_2, respectively. We consider the polynomial vector field $F : \mathbb{R}^2 \to \mathbb{R}^2$, $F(x,y) = (p(x,y), q(x,y))$. A zero z of F is a pair (x_0, y_0) of real numbers for which $F(x_0, y_0) = (0,0)$.

Let A be a rectangle in the plane defined by $a_1 \le x \le a_2$, $a_3 \le y \le a_4$, $a_1 < a_2$, $a_3 < a_4$, $a_i \in \mathbb{Q}$, $i = 1,2,3,4$ so that no zero of F lies on its boundary ∂A, and $p \cdot q$ does not vanish at its vertices. Then we can introduce the Gauss map $G : \partial A \to S^1$, $G = F/\|F\|$, where S^1 is the unit circle. Since $\|F\| \neq 0$ on ∂A, G is continuous. Assume that both ∂A and S^1 carry the counterclockwise orientation. Then the degree d of G is an integer that, roughly speaking, tells how many times ∂A is wrapped around S^1 by G. More precisely, if $H_1(\partial A)$, $H_1(S^1)$ are the first homology groups of ∂A and S^1, and $G_* : H_1(A) \to H_1(S^1)$ is the associated map, then d is the unique integer that satisfies $\beta \circ G_* \circ \alpha^{-1}(u) = d \cdot u$, for all $u \in \mathbb{Z}$, where $\alpha : H_1(\partial A) \to \mathbb{Z}$, $\beta : H_1(S^1) \to \mathbb{Z}$ are the natural isomorphisms of the homology groups and the group of integers $\mathbb{Z}$.

We will present a method for the computation of d using the Euclidean algorithm. The idea is based on a paper by Wilf [8], in which he gives a procedure that finds the complex roots of a polynomial in one complex variable. The basic tool of our computation is the notion of the Cauchy index (§2), which has been extensively used in computational problems, such as the problem of Routh and Hurwitz [2], [4]. The main result of this paper is Proposition 1 (§2).

In §3 we present an application of our procedure concerning signs of algebraic numbers as well as the topology of real algebraic curves.

We use only polynomials with integer coefficients and evaluate those polynomials at rational points. This is because the integers (rationals) are exactly representable in a computer, and therefore all calculations can be carried out exactly. The operating time of the algorithm is dominated by $n^2(n\lambda + \ell + \log n + \lambda)$ (see §4).

2. Preliminaries and the algorithm. Let z be a zero of F. Since p and q have no common factors, z is isolated. Consider a small circle $S(z,\epsilon)$ centered at z, of radius $\epsilon, \epsilon > 0$ so that

(i) z is the only zero of F inside $S(z,\epsilon)$,

*Received by the editors October 5, 1987; accepted for publication (in revised form) October 5, 1989.

†IBM Research Division, T. J. Watson Research Center, Yorktown Heights, New York 10598. Present address, Department of Mathematical Sciences, New Mexico State University, Las Cruces, New Mexico 88003.

(ii) $F \neq 0$ on $S(z,\epsilon)$.

Then we define the map $G' : S(z,\epsilon) \to S^1$, $G' = F/\|F\|$. Assume that both $S(z,\epsilon)$ and S^1 carry the counterclockwise orientation. Then we can define the degree d' of G', in exactly the same way as we did for G. It is well known that d' is independent of the particular $S(z,\epsilon)$ chosen [3].

DEFINITION 1. *With the above considerations we define the index of F at z, $i(z)$, to be d'.*

Remark 1. $d = \sum_z i(z)$, where z is a zero of F lying in the interior of A [3].

Example 1. Let $p(w)$ be a complex polynomial in the variable w. Write $p(w) = R(w) + iI(w)$, and consider $F = (R, I)$, where R, I are the real and imaginary parts of $p(w)$. Then if z is a zero of F, we observe that $i(z) =$ multiplicity of z as a root of $p(w)$. Furthermore, $d =$ the number of roots z (along with their multiplicities) that lie inside A.

Our preliminary goal is to compute d. We begin with some preliminaries.

DEFINITION 2. *Let $R(x)$ be a rational function, and $[a,b]$ a closed interval, $a < b$, so that R does not become infinite at the endpoints. By the Cauchy index, $I_a^b R$ of R over $[a,b]$, we mean $I_a^b R = N_-^+ - N_+^-$, where $N_-^+ (N_+^-)$ denotes the number of points in (a,b) at which $R(x)$ jumps from $-\infty$ to $\infty (\infty$ to $-\infty)$, respectively, as x is moving from a to b. By convention $I_a^b R = -I_b^a R$.*

According to the definition, if $R(x) = \sum_{i=1}^m A_i/(x - a_i) + R_1(x)$, where A_i, a_i, $i = 1, \cdots, m$ are real numbers and $R_1(x)$ is a rational function without real poles, then

$$I_a^b R = \sum_{a<a_i<b} \operatorname{sign} A_i \, .$$

Now let $R(x)$ be defined as $R(x) = r(x)/s(x)$, where $r(x)$, $s(x)$ are integer polynomials, and let $[a,b]$ be a closed interval as in the above definition. We will then give an algorithmic procedure for computing $I_a^b R$.

First, assume that $\deg s(x) \geq \deg r(x)$. Then, using the Euclidean algorithm for computing the greatest common divisor of two polynomials, we can find a sequence $f_1, f_2, \cdots, f_\tau$ of polynomials as follows: $f_1 = s$, $f_2 = r$, and $f_i = q_i f_{i+1} - f_{i+2}$; $\deg f_{i+2} < \deg f_{i+1}$, $i = 1, \cdots, \tau - 2$, and $f_\tau = \gcd(r, s)$.

The following theorem, due to Sturm, is the basis for the calculation of the Cauchy index $I_a^b R$.

THEOREM 1 [2]. *Let $f_1, \cdots, f_\tau$ be the sequence above. Let $V(x)$ denote the number of sign changes in the sequence of numbers $f_1(x), \cdots, f_\tau(x)$, $x \in \mathbb{R}$. Then*

$$I_a^b R = I_a^b \frac{r}{s} = V(a) - V(b) \, .$$

Now suppose that $\deg s(x) < \deg r(x)$. Then we can write $r/s = r_1/s + r_2$, where $\deg r_1 < \deg s$. We observe that $I_a^b r/s = I_a^b r_1/s$. This is because $r(x) = r_1(x) + r_2(x) \cdot s(x)$, and at zero x_0 of s we have $r(x_0) = r_1(x_0)$; that is, r and r_1 have the same sign near a zero of s.

We summarize: if $R(x) = r(x)/s(x)$, $[a,b]$ are as above, we can compute the Cauchy index $I_a^b R$ using the Euclidean algorithm and Theorem 1.

Now let $[c,c']$ be an interval so that $r(c) = r(c') = 0$, $s(c) \cdot s(c') \neq 0$ and $r(t) \neq 0$ on (c,c'), $c < c'$, where $r(x), s(x)$ are as above. Let $\sigma(t) = (s(t), r(t))$, $\sigma : [c,c'] \to \mathbb{R}^2$. We denote by $\Delta = \Delta_{c \leq t \leq c'} \arg \sigma(t)$ the change in the argument of $\sigma(t)$ as t goes from c to c'. The following lemma gives us a first connection between Δ and $I_c^{c'} r/s$.

LEMMA 1. $\Delta = -\pi \cdot I_c^{c'} r/s$.

Proof. Without loss of generality, assume that $r(t) > 0$ on (c, c'). We are going to prove the lemma by considering various cases. Suppose first that $s(c) > 0$ and $s(c') > 0$. In this case $\Delta = 0$. On the other hand, the number of points $t_0, t_0 \in (c, c')$ where $s(t)$ changes from positive to negative is equal to the number of points it changes from negative to positive; i.e., $N_+^- = N_-^+$ and, therefore, $I_c^{c'} r/s = 0$. Similarly, we treat the case where $s(c) < 0$ and $s(c') < 0$. Now if $s(c) > 0$, and $s(c') < 0$, we have $\Delta = \pi$. But the number of points at which $s(t)$ changes from positive to negative is one more than the number of points it changes from negative to positive and, therefore, $N_+^- - N_-^+ = 1$. Similarly, if $s(c) < 0$ and $s(c') > 0$, we get $N_+^- - N_-^+ = -1$. □

COROLLARY 1. *Let r, s, σ be as before and let $[\alpha, \beta]$ be a closed interval so that $r(\alpha) = r(\beta) = 0$, $s(\alpha) \cdot s(\beta) \neq 0$, $\alpha < \beta$. Then*

$$\Delta_{\alpha \leq t \leq \beta} \arg \sigma(t) = -\pi \cdot I_\alpha^\beta \frac{r}{s} .$$

Proof. Let $t_1, \cdots, t_k$ be the distinct real roots of $r(t)$ inside (α, β), $t_i < t_{i+1}$, $i = 1, \cdots, k-1$. Set $t_0 = \alpha$, $t_{k+1} = \beta$. Then

$$\Delta_{\alpha \leq t \leq \beta} \arg \sigma(t) = \sum_{i=0}^{k} \Delta_{t_i \leq t \leq t_{i+1}} \arg \sigma(t) = -\pi \cdot I_\alpha^\beta \tfrac{r}{s} . \qquad \square$$

Recall from the introduction that we are given the rectangle A, which is defined by $a_1 \leq x \leq a_2$, $a_3 \leq y \leq a_4$, and it is such that F does not vanish on its boundary, and $p \cdot q$ is not zero at each vertex of A; such an A will be called proper for F. Let $\sigma(t) : [0, 1] \to \partial A$ be an orientation-preserving parametrization of ∂A. We observe that

$$\frac{1}{2\pi} \Delta_{0 \leq t \leq 1} \arg(F \circ \sigma)(t) = d .$$

We associate an integer $I_A F$ to A and F as follows: let

$$R_3 = \frac{q(x, a_3)}{p(x, a_3)} , \; R_2 = \frac{q(a_2, y)}{p(a_2, y)} , \; R_4 = \frac{q(x, a_4)}{p(x, a_4)} , \; R_1 = \frac{q(a_1, y)}{p(a_1, y)} .$$

We then set

$$I_A F = I_{a_1}^{a_2} R_3 + I_{a_3}^{a_4} R_2 + I_{a_2}^{a_1} R_4 + I_{a_4}^{a_3} R_1 .$$

The following proposition is the main result of this paper.

PROPOSITION 1. *$I_A F$ is an even integer and, furthermore, $d = -\frac{1}{2} I_A F$.*

Proof. We have $d = \frac{1}{2\pi} \Delta_{0 \leq t \leq 1} \arg(F \circ \sigma)(t) = \frac{1}{2\pi} \cdot \pi \sum_{\partial A} (N_+^- - N_-^+) = -\frac{1}{2} I_A F$, as Corollary 1 shows. □

3. An application. I. Let F, A, G be as in the previous section. Let $J = \partial p/\partial x \; \partial q/\partial y - \partial q/\partial x \; \partial p/\partial y$ be the Jacobian determinant of F, and $z_0 = (x_0, y_0) \in \mathbb{R}^2$ be a zero of F. We say that z_0 is nondegenerate if $J(z_0) \neq 0$. Suppose that all zeros of F, which lie in the interior, Int A, of A are nondegenerate. Then Proposition 1 yields the following:

LEMMA 2. *Under the above considerations,*

$$\sum_{\substack{F(z_0) = (0,0) \\ z_0 \in \text{ Int } A}} \textit{sign } J(z_0) = -\frac{1}{2} I_A F .$$

We now proceed with a result concerning signs of algebraic numbers. Let $h(x)$, $H(x) \in \mathbb{Q}[x]$, $[a, b], a < b$ a rational interval isolating a real root x_0 of $h(x)$. We may assume that x_0 is a simple root of $h(x)$. Then by replacing H, if necessary, with $H \pm h$,

we can suppose that $H(a)H(b) \neq 0$. Our aim is to determine the sign of $H(x_0)$. First, consider $D = \gcd(h, H)$. Then for $x \in \mathbb{R}$ we set

$$V_\infty(x) = \begin{cases} 1 & \text{if } h(x) < 0 \\ 0 & \text{otherwise} \end{cases}, \quad V_0(x) = \begin{cases} 1 & \text{if } h(x)H(x) > 0 \\ 0 & \text{otherwise}, \end{cases}$$

and let I be the following integer,

$$I = V_\infty(a) - V_0(a) - I_a^b \frac{h}{H} + V_0(b) - V_\infty(b).$$

As a result, we have the following proposition.

PROPOSITION 2 [5]. (i) $H(x_0) = 0$ *if and only if* $D(a)D(b) < 0$.
(ii) *If* $D(a)D(b) \geq 0$, *then* $H(x_0) > 0$, $H(x_0) < 0$ *if and only if* $I \neq 0$, $I = 0$, *respectively.*

Proof. (ii) Let Γ be the vector field defined by $\Gamma = (h(x), y - H(x))$, and let M be a positive integer so that $\max_{a \leq x \leq b} |H(x)| < M$. Also, consider the rectangle $A = [a, b] \times [0, M]$. First, we observe that $z_0 = (x_0, H(x_0))$ is the only zero of Γ within the region $a < x < b$. Further, z_0 is nondegenerate, since x_0 is a simple root of $h(x)$. A calculation now verifies that $I = -I_A\Gamma$, and therefore as Lemma 2 shows, z_0 is inside, outside A, if and only if $I \neq 0$, $I = 0$, respectively. □

Now let us assume that all the roots of $h(x)$ are simple, and let A be a proper rectangle for $\Gamma = (h(x), y - H(x))$. Also, let m denote the number of zeros of Γ in the interior Int A, of A. The following Proposition provides a means of isolating the zeros of Γ in the interior of A.

PROPOSITION 3. *Let* Γ, A *and* m *be as above. Then,*

$$m = -\frac{1}{2} I_A \Gamma^*$$

where $\Gamma^* = (h(x), yh'(x) - h'(x)H(x))$.

Proof. We first note that z_0 is a zero of Γ^* if and only if z_0 is a zero of Γ. Furthermore, if J^* is the Jacobian determinant of Γ^*, we observe that $J^*(z_0) = (h'(x_0))^2 > 0$, where $z_0 = (x_0, H(x_0))$ is a zero of Γ^*. Therefore, $m = -\frac{1}{2} I_A \Gamma^*$, as Lemma 2 shows. □

Finally, let $c \in \mathbb{R}$ and $a, b \in [-\infty, +\infty], a < b$, so that $H(x_i) \neq c$, at each $h(x_i) = 0, a < x_i < b$, and $h(a)h(b) \neq 0$. Denote by n_a^b, k_H^c the number of roots x_i of $h(x)$ in (a, b), and the number of those x_i's at which $H(x_i) > c$, respectively.

PROPOSITION 4. *Let* h, H, n_a^b *and* k_H^c *be as above. Then,*

$$2k_H^c = n_a^b - I_a^b \frac{h'(c - H)}{h} \quad .$$

Proof. Let M be as in Proposition 2, $c < M$, and consider the rectangle $A = [a, b] \times [c, M]$. Then Proposition 3 shows that

$$-2k_H^c = I_a^b \frac{h'(c - H)}{h} + I_b^a \frac{h'(M - H)}{h} = I_a^b \frac{h'(c - H)}{h} - n_a^b \quad ,$$

since $M - H > 0$ and, $I_b^a \frac{h'(M-H)}{h} = -I_a^b \frac{h'}{h} = -n_a^b$. □

II. In this paragraph we state two results, taken from [6], that are concerned with the topology of real curves. Let $f \in \mathbb{Q}[x,y]$ be such that the coefficient of y^n is a nonzero constant, where n is the total degree of f. Let $C = \{f(x,y) = 0\}$ and suppose that C is real nonsingular. Denote by h the restriction of the projection map $(x,y) \to x$ on the curve C. Let $z_0 = (x_0, y_0) \in \mathbb{R}^2$ be a critical point of h; that is $f(z_0) = \partial f/\partial y(z_0) = 0$. Consider the vector field $F = (f, \partial f/\partial y)$ and let A be a proper rational rectangle isolating z_0. Further, let G^1, G^2, G^3 denote the graphs of $x - x_0 = (y-y_0)^{2k_1}$, $x - x_0 = -(y-y_0)^{2k_2}$, $x - x_0 = \pm(y-y_0)^{2k_3+1}$, $k_1, k_2, k_3 \in \mathbb{Z}^+$.

The following proposition provides the basis for the local topology of the curve near z_0.

PROPOSITION 5 [6]. *Let F, A, G^1, G^2, G^3 be as above. Then $I_A F = -2, 2, 0$ if and only if near z_0, C looks like G^1, G^2, G^3, respectively.*

Now let $T = (x-s)^2 + (y-t)^2$, $s,t \in \mathbb{Q}$, be such that $T|_C$ is a Morse function [3]. Consider $p(x,y) = (y-t)\partial f/\partial x - (x-s)\partial f/\partial y$ and denote by $F = (f,p)$. Let A be a rational rectangle so that all real zeros of F are inside A. Then we have Proposition 6.

PROPOSITION 6 [6]. *The number of unbounded components of C is equal to $-\frac{1}{2} I_A F$.*

We close this section with an example which was carried out using the SCRATCHPAD II Computer Algebra System.

Example 2. Let $C = \{f = 2x^4 + y^4 - 3x^2y^2 - x^2 + x + y - 1 = 0\}$. It can be shown that C is real nonsingular. Further, if $T = x^2 + y^2$, then $T|_C$ is a Morse function and all of its critical points lie inside the rectangle $A = [-2,2] \times [-2,2]$. Now consider $p(x,y) = y\,\partial f/\partial x - x\,\partial f/\partial y = 14x^3y - 10xy^3 - 2xy + y - x$, and $F = (f,p)$. A calculation shows that $I_A F = -8$, and therefore C has four unbounded components.

4. Computing time. In this section we shall study the computing time of the algorithm. We begin with some well-known notions.

Let $k \in \mathbb{Z}$, $\alpha/\beta \in \mathbb{Q}$, $(\alpha,\beta) = 1$. We define the size of k, α/β to be $\log|k|$ and $\log|\alpha| + \log|\beta|$, respectively. Next, we define the size of a nonzero polynomial $B \in \mathbb{Z}[x,y]$ to be the maximum of the sizes of its nonzero coefficients.

Having done that we arrive at the parameters of our problem. These will be $n = \max\{n_1, n_2\}$, ℓ = maximum of the sizes of $p(x,y)$ and $q(x,y)$, and λ = maximum of the sizes of the a_i's, $i = 1, \cdots, 4$.

As we saw in the preceding section, our algorithm depends solely on the computation of the Cauchy index of a rational function. We also saw that one way of computing $I_a^b R$ is by constructing a sequence of polynomials using the Euclidean algorithm. We shall now see a similar way of computing the Cauchy index that will enable us to establish a better time bound for the algorithm. We first need a definition.

If P and Q are nonzero elements of $\mathbb{Z}[x]$, a negative remainder of P and Q is a polynomial D such that for some polynomial N and some $c, d \in \mathbb{Z}$, $c > 0$, and $d < 0$

$$cP = QN + dD$$

and either $D = 0$ or $\deg(D) < \deg(Q)$.

A negative polynomial remainder sequence (n.p.r.s) is a sequence of polynomials $P_1, \cdots, P_k$, $P_{k+1} = 0$, in which P_{i+2} is a negative remainder of P_i and P_{i+1}, $1 \le i \le k-1$. If P and Q are nonzero polynomials, there always exists an n.p.r.s. with $P_1 = P$ and $P_2 = Q$.

We can now state a result, similar to Theorem 1, §2, due to Sturm, which relates n.p.r.s. and Cauchy indices.

THEOREM 2 [2]. *Let $P, Q \in \mathbb{Z}[x]$, $\deg(P) \geq \deg(Q)$ and let P_i be an* n.p.r.s. *with $P_1 = P$, $P_2 = Q$, $i = 1, \cdots, k$. Also let $[a, b]$, $a < b$, be such that $P(a)P(b) \neq 0$. For $x \in \mathbb{R}$, let $V(x)$ denote the number of sign changes in the sequence of numbers $P_1(x), \cdots, P_k(x)$. Then*

$$I_a^b \frac{Q}{P} = V(a) - V(b) .$$

Let $P, Q, P_1, \cdots, P_k$ be as above. Let $m = \deg(P)$ and δ the maximum of the sizes of P and Q. Then, evaluation of the n.p.r.s. $P_1, \cdots, P_k$, using fast arithmetic, at a rational point of size δ' takes time $O(m^2(\delta + \log m + \delta'))$ [1].

Now we are ready to give upper bounds for the computing time. We shall first find the time needed to compute the Cauchy index $I_{a_1}^{a_2} R_3$, of $R_3 = q(x, a_3)/p(x, a_3)$ over $[a_1, a_2]$. We observe that the size of $q(x, a_3)$ or $p(x, a_3)$ is $O(n\lambda + \ell)$ and their degree is $O(n)$. Therefore, the time needed to compute $I_{a_1}^{a_2} R_3$ is $O(n^2(n\lambda + \ell + \log n + \lambda))$.

Finally, the calculation of $I_A F$ also takes $O(n^2(n\lambda + \ell + \log n + \lambda))$ time.

5. Concluding remarks. It is apparent how this procedure can be made to work not only for rectangles, but to include simple closed polygons, and in general any simple closed polynomially parametrized curve. It can also serve as a sufficient condition for a polynomial vector field to have real zeros inside a finite polygonal domain. However, in trying to find a similar procedure in dimensions greater than two, one runs into some difficulties. One reason for that is that we don't have a higher-dimensional analog of Sturm's theorem, and, therefore, we have to rely on other methods, such as elimination procedures. A first attempt in this direction appears in the author's Ph.D. thesis [7]. In an upcoming paper we hope to discuss a procedure in three dimensions.

REFERENCES

[1] G. E. COLLINS AND R. LOOS, *Real zeros of polynomials*, Comput. Suppl., 4 (1982), pp. 83–94.

[2] F. R. GANTMACHER, *The Theory of Matrices*, Chelsea, New York, 1960.

[3] V. GUILLEMIN AND A. POLLACK, *Differential Topology*, Prentice-Hall, Englewood Cliffs, NJ, 1974.

[4] A. S. HOUSEHOLDER, *Bigradients and the problem of Routh and Hurwitz*, SIAM Rev., 10 (1968), pp. 56–66.

[5] T. SAKKALIS, *Signs of algebraic numbers*, Proc. Conference on Computers and Mathematics, Massachusetts Institute of Technology, Springer-Verlag, Berlin, New York, 1989, pp. 130–134.

[6] ———, *The topological configuration of a real algebraic curve*, IBM Research Report, RC-13881, International Business Machines, Yorktown Heights, NY, 1988.

[7] ———, *An algorithmic application of Morse theory to real algebraic geometry*, Ph.D. Thesis, University of Rochester, Rochester, NY, 1986.

[8] H. S. WILF, *A global bisection algorithm for computing the zeros of polynomials in the complex plane*, J. Assoc. Comput. Mach., 25 (1978), pp. 415–420.

SIAM J. COMPUT.
Vol. 19, No. 3, pp. 544–569, June 1990

012

GROUP ACTION GRAPHS AND PARALLEL ARCHITECTURES*

FRED ANNEXSTEIN†, MARC BAUMSLAG†, AND ARNOLD L. ROSENBERG‡

Abstract. The authors develop an algebraic framework that exposes the structural kinship among the *deBruijn, shuffle-exchange, butterfly*, and *cube-connected cycles* networks and illustrate algorithmic benefits that ensue from the exposed relationships. The framework builds on two algebraically specified genres of graphs: A *group action graph* (GAG, for short) is given by a set V of vertices and a set Π of permutations of V: For each $v \in V$ and each $\pi \in \Pi$, there is an arc labeled π from vertex v to vertex $v\pi$. A *Cayley graph* is a GAG (V, Π), where V is the group $\mathbf{Gr}(\Pi)$ generated by Π and where each $\pi \in \Pi$ acts on each $g \in \mathbf{Gr}(\Pi)$ by right multiplication. The graphs $(\mathbf{Gr}(\Pi), \Pi)$ and (V, Π) are called *associated* graphs. It is shown that every GAG is a quotient graph of its associated Cayley graph. By applying such general results, the authors determine the following:

- The butterfly network (a Cayley graph) and the deBruijn network (a GAG) are associated graphs.
- The cube-connected cycles network (a Cayley graph) and the shuffle-exchange network (a GAG) are associated graphs.
- The order-n instance of both the butterfly and the cube-connected cycles share the same underlying group, but have slightly different generator sets Π.

By analyzing these algebraic results, it is delimited, for any Cayley graph $\mathcal{G}$ and associated GAG $\mathcal{H}$, a family of "leveled" algorithms which run as efficiently on $\mathcal{H}$ as they do on (the much larger) $\mathcal{G}$. Further analysis of the results yields new, surprisingly efficient simulations by the shuffle-oriented networks (the shuffle-exchange and deBruijn networks) of *like-sized* butterfly-oriented networks (the butterfly and cube-connected cycles networks):

- An N-vertex butterfly-oriented network can be simulated by the smallest shuffle-oriented network that is big enough to hold it with slowdown $O(\log \log N)$.

This simulation is *exponentially faster* than the anticipated logarithmic slowdown. The mappings that underlie the simulation can be computed in linear time; and they afford one an algorithmic technique for translating any program developed for a butterfly-oriented architecture into an equivalent program for a shuffle-oriented architecture, the latter program incurring only the indicated slowdown factor.

Key words. butterfly network, Cayley graph, cube-connected cycles network, deBruijn network, graph embedding, group action graph, interconnection network, network simulation, parallel architecture, shuffle-exchange network

AMS(MOS) subject classifications. 94C15, 68C25, 05C99, 68E10

1. Motivation and synopsis.

We develop an algebraic setting for studying certain structural and algorithmic properties of the interconnection networks that underlie parallel architectures. We apply the structure-oriented results derived within this setting to obtain a simulation of butterfly-oriented interconnection networks on like-sized shuffle-oriented interconnection networks, which is exponentially faster than previous simulations. Our study and approach find their motivations in three sources.

1.1. Uniform vs. semi-uniform networks.

Our first motivation relates to a paradigm advocated by Akers and Krishnamurthy [1]-[3]; Carlsson et al. [10],[9]; Faber [14]; and others, concerning how to design the interconnection network of a par-

* Received by the editors January 20, 1988; accepted for publication (in revised form) September 15, 1989. A portion of this research was supported by National Science Foundation grants DCI-87-96236 and CCR-88-12567.

† Department of Computer and Information Science, University of Massachusetts, Amherst, Massachusetts 01003.

‡ A portion of this research was done while this author was visiting the Department of Computer Science, the Technion Haifa, Israel.

allel architecture.[1] The cited sources argue that interconnection networks based on *Cayley graphs* [2](CAGs), i.e., graphs whose adjacency structure is governed by a group, endow an architecture with substantive advantages, in terms of algorithmic efficiency and fault tolerance. They support their case in part by noting that many interconnection networks of algorithmic and commercial importance are Cayley graphs, including the *hypercube* [26], [28]; *butterfly* (with wraparound) [5]; *cube-connected cycles* [9],[22]; *multiple rings* [30]; and *star* [2], [3] networks. Two factors detract from their arguments: First, there are several "semi-uniform" interconnection networks, which are promoted as vigorously by their proponents as are Cayley-graph networks by theirs; included here are the *shuffle-exchange* (and its variants) [27],[29]; *deBruijn* [21]; and *X-tree* [13] networks. Second, a well-known group-theoretic result [8] suggests that one "natural" way of defining CAG-based interconnection networks — indeed, the way used in [2] to define the star and *pancake* networks — is almost certain to yield networks that are truly enormous.[3] The first motivation for the current research was to determine the extent to which one might enjoy the benefits of Cayley graphs using smaller, "semi-uniform" networks. We have achieved this goal, to some extent, using as our formal notion of "semi-uniform" network the class of *group action graphs* (GAGs), which we define in §2 and develop in §3; indeed, two of our exemplary "semi-uniform" networks, the deBruijn and shuffle-exchange, are GAGs. We shall show that every Cayley graph $\mathcal{G}$ has an "associated" GAG, which can emulate $\mathcal{G}$'s communication capabilities with no time loss, on a large class of computations. Work still in progress [4] extends the research reported here by showing that, if the group underlying $\mathcal{G}$ has a variety of subgroups, then there exist *work-preserving processor-time trade-offs* for any parallel architecture based on $\mathcal{G}$.

1.2. The structure of hypercube-derivative networks. Our second motivation arose from the following scenario. The major shortcoming of the (Boolean) hypercube as an interconnection network is its high vertex-degrees.[4] This fact has led to the introduction of several bounded-degree *butterfly-oriented* "approximations" of the hypercube, most notably the butterfly and CCC (cube-connected cycles) networks. Ad hoc transformations of these large[5] networks have led to the smaller *shuffle-oriented* hypercube-derivative networks, most notably the shuffle-exchange and deBruijn networks, which share the size of the hypercube and which afford one computational efficiency (roughly) equal to that of the butterfly and CCC on certain important computational tasks. Indeed, many in the parallel architecture community will attest to the computational equivalence of the four networks just mentioned, supporting their assertion by exhibiting a class of parallel algorithms that can be computed with roughly equal efficiency on all hypercube-derivative networks that share certain structural characteristics (as do the four networks under discussion). Roughly speaking, the processors of these networks have addresses containing length-n bit-strings. The algorithms in the class have the property that, for each time step i, all communication is between processors whose addresses differ only in the ith place in their bit-string (but possibly differ in other components of the address as well). Many well-known,

[1] Typically, one views an interconnection network as a graph whose vertices are the processors of the architecture and whose arcs are the interprocessor communication links.

[2] Technical definitions appear in§2.

[3] The groups underlying these Cayley graphs have large *symmetric* or *alternating* groups as subgroups. Hence they have factorially many vertices.

[4] Each vertex of the N-vertex hypercube has degree $\log N$.

[5] Each uses $N \cdot \log N$ vertices to simulate the N-vertex hypercube.

efficient algorithms reside in this class, for such tasks as sorting, computing convolutions, and matrix operations; see [22], [31] for details. The second goal of the current research was to find a mathematical framework that would expose structural and algorithmic relationships among these four interconnection networks. We expected such a framework to (1) allow us to explain rigorously the asserted "equivalence" of these networks on this class of algorithms, and (2) to serve as a benchmark for the goals outlined in the previous subsection (i.e., our first motivation). Our general framework is developed in §3; its specialization to these four networks appears in §4. The framework allows us to explain the kinship among the networks by showing that they all derive their structure (in a sense made precise in §3) from the same underlying group.

> **Aside:** To whet the reader's appetite for the conclusions of §4: Figs. 1 and 2 depict the order-3 deBruijn and butterfly networks, respectively. It is not clear at first blush how to map the butterfly network onto the deBruijn network in a structure-preserving manner. Figure 3 depicts schematically how our framework accomplishes this.

1.3. The computational power of hypercube-derivatives. Our third motivation grows out of our second. We wish to extend the structural comparison described in the preceding subsection to a comparison of the communication powers of the four networks *on general computations.* To this end, we switch from our algebraic (structure-oriented) framework to the graph-theoretic (computation-oriented) framework presented in §2.3. In that section, we develop a rather demanding formal notion of one interconnection network simulating another. Other studies have used the same framework to compare other pairs of networks: In [6], hypercubes simulate like-sized complete binary tree networks with only constant slowdown; in [16], hypercubes simulate like-sized butterfly-oriented networks with at most constant slowdown; in [16], hypercubes efficiently simulate meshes; and in [5], butterfly networks simulate complete binary tree networks with constant slowdown, X-trees with doubly logarithmic slowdown, and meshes with logarithmic slowdown, all simulations being optimal.

In §5, we exploit the insights generated by the algebraic development in §§3 and 4, within the graph-theoretic framework of §2.3, to develop a technique for simulating a butterfly-oriented network on a *like-sized* shuffle-oriented network, with slowdown which is only *doubly logarithmic* in the size of the simulated network; this is exponentially faster than previous simulations.

A goal similar to ours motivates [28] (which concentrates on evaluating competitors of the hypercube) and [15] (which concentrates on comparing the shuffle-exchange network with the hypercube). The comparison in [28] is purely qualitative, hence only peripherally related to our study. The comparison in [15] focuses on a notion of the "cost-effectiveness" of a network, which is measured by combining topological properties like diameter, bisection width, and degree of connectivity. In contrast, we are aiming for a more computation-oriented comparison.

The results we report on here represent progress toward all three of our motivating goals. We find this progress particularly satisfying, since each step in our development is dictated by the inherent structure of the networks studied, as exposed by their algebraic specifications.

We view the development here as a step towards exploring the coupling of graph-theoretic techniques with group- and monoid-theoretic techniques, in the design and analysis of parallel architectures. It is hard to argue that the insights gleaned from our algebraic formulation cannot be derived from a more standard graph-theoretic

treatment. However, it is clear that the mapping that yields our simulation in §5 derives in a natural way from the coset structure exposed in the algebraic study of Part I of the paper. For us, at least, the algebraic framework has yielded a new, exploitable insight.

2. The formal setting.

2.1. Graph-theoretic notions. We deal with three levels of graph structure.

- An *undirected graph* $\mathcal{G}$ is given by a set V of *vertices* and a set E of doubleton subsets of V called *edges*. The vertices in an edge are said to be *adjacent* in $\mathcal{G}$. A *path* in $\mathcal{G}$ from vertex u to vertex v is a sequence π of vertices

$$u = v_0, v_1, \cdots, v_{\ell-1}, v_\ell = v$$

  such that, for each $0 \leq i < \ell$, $(v_i, v_{i+1}) \in E$; we say that the path π has *length* ℓ. By abuse of notation, we write "$(v_i, v_{i+1}) \in \pi$". $\mathcal{G}$ is *connected* if each pair of vertices in V appear on some path in $\mathcal{G}$.
- A *directed graph* (*digraph*, for short) $\mathcal{G}$ is given by a set V of *vertices* and a *multi*-subset of $V \times V$ called *arcs*.[6] The underlying undirected graph of $\mathcal{G}$ is obtained by replacing each arc of $\mathcal{G}$ by the corresponding unordered set and disposing of the set when it is not a doubleton. $\mathcal{G}$ is *connected* when its underlying undirected graph is. $\mathcal{G}$ is *strongly connected* if, for every ordered pair $\langle v, w \rangle$ of vertices of $\mathcal{G}$, there is a *directed path from v to w*, i.e., a sequence of vertices, $v = v_1, v_2, \cdots, v_\ell = w$, with each pair $\langle v_i, v_{i+1} \rangle$ an arc of $\mathcal{G}$.
- A *transformation graph* (TRAG, for short) is given by a set V of *vertices* and a set Φ of transformations of V. For each $v \in V$ and each $\phi \in \Phi$, there is an arc labeled ϕ from vertex v to vertex $v\phi$.[7] The digraph underlying the TRAG $\mathcal{G}$ is obtained by erasing the labels from the arcs of $\mathcal{G}$ and removing any resulting parallel arcs. A TRAG is connected (respectively, strongly connected) just when its underlying digraph is.

TRAGs are known in semigroup theory as *operands* [11]; the third author studied strongly connected TRAGs extensively, under the name *data graphs* ([23], [24] are the sources most relevant to the present study).

We now define the specific genres of graphs that occupy our attention.

A *group action graph* (GAG, for short) is a TRAG (V, Φ) for which each transformation in the set Φ is a *permutation* of the set V: For mnemonic emphasis, we henceforth denote the transformation-set Π.

Clearly, every GAG has underlying it a digraph all of whose indegrees and outdegrees[8] are equal. The proof in [17] that every such "two-way regular" digraph can be 2-factorized (cf. [7]) shows that this property is sufficient also, in that it can be used to prove the following.

PROPOSITION 2.1. *A digraph $\mathcal{G}$ admits an arc-labeling that makes $\mathcal{G}$ a* GAG *if, and only if, there is a constant c such that every vertex of $\mathcal{G}$ has both indegree c and outdegree c.*

A little background on groups is necessary before we begin to discuss the highly uniform graphs on which our study focuses.

---

[6] By using *multi*-subsets of $V \times V$, we allow "parallel" arcs, i.e., several arcs connecting a given pair of vertices.

[7] $v\phi$ denotes the image of v under the transformation ϕ.

[8] The *indegree* (respectively, *outdegree*) of a vertex v of $\mathcal{G}$ is the number of arcs of $\mathcal{G}$ having v as their second (respectively, their first) component.

2.2. Group-theoretic notions. In order to simplify exposition, we present certain notions here in a nonstandard way.

A *group* is given by a set S, together with an associative binary *multiplication* on S (denoted by a centered dot) that has an *identity* — an $e \in S$ for which $s \cdot e = e \cdot s = s$, for all $s \in S$ — and *inverses* — for each $s \in S$, an element $t \in S$ for which $s \cdot t = t \cdot s = e$.

A *Cayley graph* (or, *group graph*) is a GAG (V, Π), where V is the group $\mathbf{Gr}(\Pi)$ generated by Π, and where each $\pi \in \Pi$ acts on $\mathbf{Gr}(\Pi)$ by right multiplication, so that $\pi \in \Pi$ "leads" vertex $g \in \mathbf{Gr}(\Pi)$ to vertex $g \cdot \pi$. We call Π the set of *generators* of the group $\mathbf{Gr}(\Pi)$. We denote by $\mathbf{Cay}(\Pi)$ the Cayley graph $(\mathbf{Gr}(\Pi), \Pi)$ induced by the set Π of permutations.

There is clearly a "natural" way to construct a Cayley graph from any GAG (V, Π), namely, using the group $\mathbf{Gr}(\Pi)$, thereby obtaining $\mathbf{Cay}(\Pi)$. Less obviously, there is a "natural" way to construct GAGs (usually more than one) from any Cayley graph.

Given any subgroup H of a group G (i.e., a subset of G which is a group under the multiplication in G), the *quotient* of G by H, denoted G/H, is the collection of all *right cosets of H in G*: For each $g \in G$, the *right coset of H containing g*, denoted Hg, is the set of all left multiples $h \cdot g$ of g by elements $h \in H$. It is a standard result that the cosets of H *partition* G into blocks of equal size.

These notions yield the "natural" construction of GAGs from Cayley graphs.

Let H be a subgroup of the group $G = \mathbf{Gr}(\Pi)$. The *coset graph of G with respect to H and* Π, denoted $\mathbf{Cos}(G; H; \Pi)$, is the GAG $(G/H, \Pi)$ whose vertex-set is the set of right cosets of H in G, and whose arcs are given by the action of the elements of Π viewed as permutations of G/H. This action is defined by right multiplication: For $g \in G$ and $\pi \in \Pi$, $(Hg)\pi = H(g \cdot \pi)$.

Our study builds on the fact that a group can be viewed both as an abstract algebraic structure and as a collection of permutations — multiplication in the latter view being functional composition. This important fact is formalized in Cayley's theorem, which follows.

PROPOSITION 2.2 [20]. *Every group is isomorphic to a group of permutations.*
Let G be a group of permutations of the set S.

- *G acts transitively on S (is transitive*, for short) if, for all $s, t \in S$, there is a $g \in G$ such that $sg = t$.
- For each $s \in S$, the *stabilizer* of s in G, denoted $\mathbf{St}(G; s)$, is the set of all permutations in G that *fix* s, i.e., for which $s\pi = s$. It is a standard result that $\mathbf{St}(G; s)$ is a subgroup of G.

A *cyclic group* is a group whose underlying set is (isomorphic to) $Z_d =_{\text{def}} \{0, 1, \cdots, d-1\}$ and whose multiplication is (isomorphic to) addition modulo d. We denote the d-element cyclic group by Z_d, allowing context to distinguish between the group and its underlying set.

The *wreath product* of cyclic group Z_d by cyclic group Z_n, denoted $Z_d \otimes_{wr} Z_n$, is a group of permutations of the set

$$Z_d^n =_{\text{def}} \{0, 1, \cdots, d-1\}^n.$$

Each element of $Z_d \otimes_{wr} Z_n$ is an $(n+1)$-tuple

$$\pi = \langle \alpha; \beta_0, \beta_1, \cdots, \beta_{n-1} \rangle,$$

where $\alpha \in Z_n$ and each $\beta_i \in Z_d$. The action of the permutation π on the element $(\delta_0, \delta_1, \cdots, \delta_{n-1}) \in Z_d^n$ consists of a modulo-d vector addition of $(\beta_0, \beta_1, \cdots, \beta_{n-1})$, followed by a sequence of α left-cyclic shifts:

$$(\delta_0, \delta_1, \cdots, \delta_{n-1})\langle \alpha; \beta_0, \beta_1, \cdots, \beta_{n-1}\rangle$$

$$= (\delta_\alpha + \beta_\alpha, \cdots, \delta_{n-1} + \beta_{n-1}, \delta_0 + \beta_0, \cdots, \delta_{\alpha-1} + \beta_{\alpha-1}).$$

Multiplication in a wreath product is composition of permutations.

2.3. Processor arrays, graph embeddings, and simulations. We represent parallel architectures — really, arrays of identical processing elements (PEs) and their underlying interconnection networks — as undirected graphs in the following way. The **vertices** of the graph represent the **PEs** of the array, and the **edges** of the graph represent the inter-PE **communication links**. In order to make do with a simple notion of simulation, we assume a *pulsed* model of computation, wherein *computation* steps alternate with (point-to-point) *communication* steps between adjacent PEs; this assumption is most appropriate when the processor array operates under a SIMD (single instruction stream, multiple data stream) *regimen*, i.e., a regimen wherein, at each time step, all PEs execute the same operation.

We build on the above representation to use graph embeddings to represent the simulation of one processor array $\mathcal{A}$ by another array $\mathcal{B}$. We assume that the PEs of the simulating array $\mathcal{B}$ are sufficiently numerous and sufficiently powerful to simulate the PEs of the simulated array $\mathcal{A}$ step for step — so no delay is incurred because of computational steps. We restrict attention to simulations that honor the pulsed computation regimen of array $\mathcal{A}$: Array $\mathcal{B}$ alternates (single) steps that simulate one *computation* step of array $\mathcal{A}$, with (possibly multiple) steps that simulate one *communication* step of array $\mathcal{A}$. The slowdown incurred by a simulation arises from having to simulate on the interconnection network underlying array $\mathcal{B}$, communication steps that are tailored to the (possibly very different) structure of the interconnection network underlying array $\mathcal{A}$. This delay results both from mismatched adjacency structures and from congested communication channels. Our formal notion of simulation resides in the following notion of graph embedding. An *embedding* of the graph $\mathcal{G}$ in the graph $\mathcal{H}$ is specified by

- a one-to-one *assignment* α of the vertices of $\mathcal{G}$ to the vertices of $\mathcal{H}$:

$$\alpha : V_{\mathcal{G}} \to V_{\mathcal{H}}.$$

- a *routing* ρ of each edge (u, v) of $\mathcal{G}$ along a distinct path in $\mathcal{H}$ connecting vertices $\alpha(u)$ and $\alpha(v)$:

$$\rho : E_{\mathcal{G}} \to \text{Paths}(\mathcal{H}).$$

Fundamental to our representing simulations by graph embeddings is our assessing the *delay* incurred by a simulation. We use three measures for this purpose. Say that we have an embedding (α, ρ) of $\mathcal{G}$ in $\mathcal{H}$.

- The *dilation* of the embedding is the maximum amount that the routing ρ "stretches" any edge of $\mathcal{G}$:

$$\text{dilation}(\alpha, \rho) = \max_{(u,v)\in E_{\mathcal{G}}} \text{Length}(\rho(u, v)).$$

- The (*edge*)*congestion* of the embedding is the maximum number of edges of $\mathcal{G}$ that ρ routes over a single edge of $\mathcal{H}$:

$$\text{congestion}(\alpha, \rho) = \max_{e\in E_{\mathcal{H}}} |\{e' \in E_{\mathcal{G}} \ : \ e \in \rho(e')\}|.$$

- The *dynamic* (edge) *congestion* of the embedding is the maximum number of atomic messages that must be transmitted over any edge of $\mathcal{H}$ *simultaneously,* when $\mathcal{H}$ simulates a single communication step of $\mathcal{G}$ within our pulsed simulation regimen.

Both congestion and dynamic congestion measure contention for communication links, which can be resolved either by increasing the bandwidth of the links, at the cost of increased hardware and increased area, or via queuing of messages, at the cost of increased delay. For the simulations presented here, dynamic congestion will be bounded by $O(1)$, while congestion will be bounded only by $O(\log \log N)$; therefore, one might opt here to handle dynamic congestion via increased bandwidth and congestion via message queuing.

The reader can verify the following simplifying inequalities.

LEMMA 2.3. *For any embedding* (α, ρ) *of a graph* $\mathcal{G}$ *into a graph* $\mathcal{H}$ *whose maximum vertex-degree is* d,

$$\frac{\text{congestion}(\alpha,\rho)}{\text{dilation}(\alpha,\rho)} \leq \text{dynamic congestion}(\alpha,\rho) \leq \text{congestion}(\alpha,\rho) < d^{\text{dilation}(\alpha,\rho)}.$$

Our primary interest here is algorithmic — we want to determine how efficiently one architecture $\mathcal{H}$ can simulate another architecture $\mathcal{G}$ on general computations — but, our formal setting is purely graph-theoretic. The small dynamic congestion of our embeddings renders transparent the translation from our graph-theoretic parameters to a computational measure of slowdown. In more general situations, one must invoke a recent result from [18] to see that the purely graph-theoretic notion of simulation outlined here captures the essence of the algorithmic problem, in the following precise sense.

PROPOSITION 2.4. [18]. *Say that one can embed the graph* $\mathcal{G}$ *in the graph* $\mathcal{H}$, *with dilation* D *and congestion* C. *Then the architecture* $\mathcal{H}$ *can simulate* T *steps of the architecture* $\mathcal{G}$ *on a general computation in* $O(C + D)T$ *steps.*

Part I: Algebraic development.

3. Algebraic results: Abstract version.

3.1. GAGs and coset graphs. There is often a close relationship between the structure of the group underlying a GAG $\mathcal{G}$ and certain types of uniformity in the (unlabeled) graph underlying $\mathcal{G}$.

LEMMA 3.1. *Every connected* GAG *is strongly connected. It follows that, if* (V, Π) *is a connected* GAG, *then* $\mathbf{Gr}(\Pi)$ *is a transitive group.*

Proof. By Proposition 2.1, the digraph underlying a connected GAG is a connected digraph with equal indegree and outdegree at each vertex. It is well known (cf.[7]) that every such digraph contains an *Eulerian tour* (i.e., a closed directed path traversing each arc exactly once). This tour yields a directed path between any pair of vertices, hence the claim of strong connectivity. To see that $\mathbf{Gr}(\Pi)$ is transitive, observe that for any pair of vertices (v, w), the sequence of labels on a directed path from v to w defines (via composition of permutations) a permutation in $\mathbf{Gr}(\Pi)$ that maps v to w. □

We are now in a position to prove that every connected GAG $\mathcal{G} = (V, \Pi)$ is a coset graph. This structure theorem is one in a large collection of similar results in a variety of areas of mathematics, including differential geometry and the theory of topological groups.

Since the group $G = \mathbf{Gr}(\Pi)$ is a transitive permutation group (by Lemma 3.1), we can simplify our structure theorem slightly. We are concerned with stabilizers of

elements of V in G. In a transitive permutation group, all stabilizers are conjugate[9]; hence all are isomorphic. Thus, we can refer to *the stabilizer subgroup of* G, written $\mathbf{St}(G)$, without focusing on which $v \in V$ we are fixing.

THEOREM 3.2. *Every connected* GAG (V, Π) *is isomorphic* (as a TRAG) *to the coset graph*

$$\mathbf{Cos}(\mathbf{Gr}(\Pi);\ \mathbf{St}(\mathbf{Gr}(\Pi));\ \Pi).$$

Proof. Let the GAG $\mathcal{G} = (V, \Pi)$ and the group $G = \mathbf{Gr}(\Pi)$ be as in the statement of the theorem. Pick an arbitrary $v \in V$,[10] and let $H = \mathbf{St}(G; v)$. Establish the following mapping μ from cosets in G/H to V: for $g \in G$,

$$(Hg)\mu = w \text{ if and only if } (vh)g = v(h \cdot g) = w \text{ for each } h \in H$$

μ thus associates vertex w of $\mathcal{G}$ with that coset of G/H which comprises the permutations in G that map v to w. Since G is a group of permutations, and since H is the stabilizer of v in G, the mapping μ is well defined and one-to-one; moreover, since G is transitive (by Lemma 3.1), the mapping μ is also onto. Therefore, once we show that μ preserves (labeled) arcs, we shall be done.

Say first that, in the GAG, there is an arc labeled π from vertex u to vertex w. By definition, π is a permutation in Π for which $u\pi = w$. Now, for every permutation $g \in G$ that maps v to u (i.e., $vg = u$), the permutation $g \cdot \pi$ maps v to $u\pi = w$, via the equations

$$w = u\pi = (vg)\pi = v(g \cdot \pi).$$

It follows that there is an arc labeled π from vertex $u\mu$ to vertex $w\mu$ in the coset graph.

Finally, say that, in the coset graph, there is an arc labeled π from vertex Hf to vertex Hg $(f, g \in G)$. By definition, we have the equation

$$Hg = H(f \cdot \pi)$$

on the right cosets of H. It follows that for each permutation $h \in H$, we have

$$v(h \cdot g) = (vh)g = vg = v(f \cdot \pi) = (vf)\pi$$

since H is the stabilizer of v in G. But the latter equations imply that there is an arc labeled π from vertex vf to vertex vg in the GAG. □

Although the choice of the stabilizer H in Theorem 3.1 does not affect the structure of the GAG, it does change the correspondence between the right cosets of H in G and vertices of $\mathcal{G}$.

3.2. Structural consequences of Theorem 3.1. We turn now to three corollaries of Theorem 3.1 that exhibit useful relationships between a GAG $\mathcal{G} = (V, \Pi)$ and its induced Cayley graph $\mathbf{Cay}(\Pi)$.

A. Replication of substructures. Our first corollary indicates that certain simple structures in $\mathcal{G}$ replicate in $\mathbf{Cay}(\Pi)$ with the multiplicity suggested by Theorem 3.1's characterization of GAGs as coset graphs. The proof of this result greatly simplifies a purely combinatorial proof in [5]; cf. Corollary 4.5.

[9] Subgroups H_1 and H_2 of G are *conjugate* if every right coset of H_1 by any $g \in G$ is a left coset of H_2 by the same element; symbolically, $H_1 g = gH_2$ for all $g \in G$.

[10] The specific v chosen is immaterial, since G is transitive.

COROLLARY 3.3. *Each directed tree*[11] $\mathcal{T}$ *that is a subgraph of the* GAG (V, Π) *appears as a subgraph of* $\mathbf{Cay}(\Pi)$ *with multiplicity* $|\mathbf{St}(\mathbf{Gr}(\Pi))|$.

Proof. Let the directed tree $\mathcal{T}$ be a subgraph of the GAG $\mathcal{G} = (V, \Pi)$. The mapping μ in the proof of Theorem 3.1 associates each vertex v of $\mathcal{T}$ with a unique right coset Hg for some $g \in \mathbf{Gr}(\Pi)$, where $H = \mathbf{St}(\mathbf{Gr}(\Pi); w)$ for some fixed but arbitrary $w \in V$. As we noted earlier, all of the candidate cosets have common size $|\mathbf{St}(\mathbf{Gr}(\Pi))|$.

Consider now an arbitrary arc labeled $\pi \in \Pi$ from vertex u to vertex w in $\mathcal{G}$. In $\mathbf{Cay}(\Pi)$, the permutation π induces a bijection (via right multiplication) between the cosets $\mu^{-1}(u)$ and $\mu^{-1}(w)$. Thus, each arc in $\mathcal{T}$ spawns $|\mathbf{St}(\mathbf{Gr}(\Pi))|$ distinct arcs in $\mathbf{Cay}(\Pi)$. Since no two arcs in the tree $\mathcal{T}$ enter the same vertex, this replication of arcs suffices to establish the result. □

Note that one cannot extend Corollary 3.3 very far, since a cycle in the GAG $\mathcal{G}$ may not result in a cycle in the induced Cayley graph: the "initial" and "final" arcs in the Cayley graph will start and end (respectively) in the same coset, but not necessarily at the same vertex in the coset.

The next two corollaries of Theorem 3.1 build on the following general technique for decomposing the elements of a group. Let $G = \mathbf{Gr}(\Pi)$ be a group, and let H be a subgroup of G. We can establish a bijection

$$\tau : G \longrightarrow H \times G/H$$

in the following way. In the coset graph $\mathbf{Cos}(G; H; \Pi)$, fix a *breadth-first* spanning tree $\mathcal{T}$, rooted at vertex H. By Corollary 3.3, there is a spanning forest of $\mathbf{Cay}(\Pi)$ consisting of copies of $\mathcal{T}$ rooted at each element of H. Associate each element $g \in \mathbf{Gr}(\Pi)$ with that element $h_g \in H$ that is the root of the tree containing g. Extend this association to the desired bijection via the rule $g\tau = \langle h_g, Hg \rangle$. This mapping τ is one-to-one because elements of the same coset Hg reside in distinct trees, hence have different h_g's; the mapping is onto because the sets G and $H \times G/H$ are equinumerous. When the subgroup H is generated by a subset Ψ of Π, one can show that this bijection gives us an embedding of the Cayley graph $\mathbf{Cay}(\Pi)$ in the *product graph*[12] $\mathbf{Cay}(\Psi) \times \mathbf{Cos}(G; H; \Pi)$.

B. Routing and diameters in Cayley graphs. Theorem 3.1 allows us to derive a scheme for point-to-point routing in a Cayley graph from a similar scheme in an associated GAG[13]. The derived routings are often optimal; and they afford us a general upper bound on the diameter[14] of a Cayley graph, in terms of the diameter of an associated GAG.

Note first that, because of the symmetry of a Cayley graph $\mathcal{G} = \mathbf{Cay}(\Pi)$, we lose no generality by restricting attention to optimal routings from the "identity" vertex e (the identity of $\mathbf{Gr}(\Pi)$) to the other vertices of $\mathcal{G}$. To wit, the shortest path from vertex u to vertex v in $\mathcal{G}$ follows the same sequence of arc-labels (i.e., group-generators) as does the shortest path from vertex e to vertex $u^{-1}v$.

---

[11] A *tree* is a connected undirected graph that has a unique path between every pair of vertices; a *leaf* in the tree is a vertex of unit degree. A *directed tree* is a digraph whose underlying graph is a tree with a designated *root* vertex, all of whose arcs are oriented from root to leaf.

[12] Given graphs $\mathcal{G} = (V_\mathcal{G}, E_\mathcal{G})$ and $\mathcal{H} = (V_\mathcal{H}, E_\mathcal{H})$, the *product graph* $\mathcal{G} \times \mathcal{H}$ has vertex-set $V_\mathcal{G} \times V_\mathcal{H}$. For $u, v \in V_\mathcal{G}$ and $x, y \in V_\mathcal{H}$, the pair $(\langle u, x \rangle, \langle v, y \rangle)$ is an edge of $\mathcal{G} \times \mathcal{H}$ just when either $(u, v) \in E_\mathcal{G}$ and $x = y$ or $(x, y) \in E_\mathcal{H}$ and $u = v$.

[13] We call the Cayley graph $\mathbf{Cay}(\Pi)$ and the GAG (V, Π) *associated* graphs.

[14] The *distance* between vertices v and w of a digraph $\mathcal{G}$ is the length of the shortest directed path from v to w; the *diameter* of $\mathcal{G}$, denoted $\mathrm{Diam}(\mathcal{G})$, is the maximum intervertex distance in $\mathcal{G}$.

Next, note that we can find a path between vertex e and any other vertex $g \in \mathbf{Gr}(\Pi)$, using our decomposition of $\mathbf{Gr}(\Pi)$: We perform our decomposition of $\mathbf{Gr}(\Pi)$, using vertex/element e as the root of one of the breadth-first trees in our spanning forest. Then

1. We find a path from root-vertex e to the root h_g of the breadth-first tree $\mathcal{T}_g$ that contains vertex/element g.
2. We find a path from root-vertex h_g to g.

If we seek shortest paths in each of the two stages of this routing, then

1. The first leg of the routing will have distance no greater than the maximum distance in $\mathcal{G}$ between any two elements of H; we denote this quantity by $\mathrm{Diam}_H(\mathcal{G})$. If the subgroup H is generated by the subset Ψ of Π, then $\mathrm{Diam}(\mathbf{Cay}(\Psi))$ is also an upper bound to this maximum distance.
2. The second leg of the routing will have distance no greater than $\mathrm{Diam}(\mathbf{Cos}(G; H; \Pi))$. This is guaranteed by the fact that $\mathcal{T}_g$ is a *breadth-first* spanning tree of $\mathbf{Cos}(G; H; \Pi)$.

We summarize the quantitative aspects of this discussion as follows.

COROLLARY 3.4. *Let $\mathcal{F} = (V, \Pi)$ be a connected* GAG, *with associated Cayley graph $\mathcal{G} = \mathbf{Cay}(\Pi)$. Letting $H = \mathbf{St}(\mathbf{Gr}(\Pi))$, we have*

$$\mathrm{Diam}(\mathcal{G}) \leq \mathrm{Diam}_H(\mathcal{G}) + \mathrm{Diam}(\mathcal{F}).$$

If H is generated by a subset Ψ of Π, then also

$$\mathrm{Diam}(\mathcal{G}) \leq \mathrm{Diam}(\mathbf{Cay}(\Psi)) + \mathrm{Diam}(\mathcal{F}).$$

C. Cayley graph bisection bounds. An *edge-bisector* (respectively, *vertex-bisector*) for a graph $\mathcal{G}$ is a set of edges (respectively, of vertices) whose removal partitions $\mathcal{G}$ into two subgraphs with equal numbers of vertices (within 1). Theorem 3.1 allows us to bound the size of the smallest edge- and vertex-bisectors of a Cayley graph in terms of the corresponding quantity of an associated GAG.

COROLLARY 3.5. *If the* GAG $\mathcal{G} = \mathbf{Cos}(\mathbf{Gr}(\Pi); H; \Pi)$ *has an edge-bisector (respectively, a vertex-bisector) of size n, then the Cayley graph $\mathcal{F} = \mathbf{Cay}(\Pi)$ has an edge-bisector (respectively, a vertex-bisector) of size $n \cdot |H|$.*

Proof. We argue only about edge-bisectors, the proof for vertex-bisectors being analogous. Say we are given an edge-bisector B for $\mathcal{G}$. Each edge in B corresponds naturally to $|H|$ edges in $\mathcal{F}$, one edge for each element of the coset from which the edge "emanates." Let B' be the set of $|H| \cdot |B|$ edges of $\mathcal{F}$ that correspond to all of the edges in B. We claim that B' is an edge-bisector of $\mathcal{F}$ (which will prove the result, since it has the right cardinality). To see this, partition the vertices of $\mathcal{F}$ according to the edge-bisection of $\mathcal{G}$ effected by B: two vertices of $\mathcal{F}$ reside in the same block of the partition if, and only if, the cosets of H to which they belong reside in the same block of the partition of $\mathcal{G}$. We claim that this partition of $\mathcal{F}$ is the edge-bisection that is effected by removing the edges in B' from $\mathcal{F}$. To wit, every edge (u, v) of $\mathcal{F}$ engenders an edge (Hu, Hv) in $\mathcal{G}$ (by definition of coset graph). Therefore, if the edge (Hu, Hv) is a bisector edge, i.e., an element of B, then the edge (u, v) must be an element of B'. The result follows. □

3.3. Algorithmic consequences of Theorem 3.1. The development in §3.1 has algorithmic consequences that often allow one to trade significant savings in the number of processors in one's array for a modest increase in computing time. For a

special class of algorithms, which includes certain algorithms for sorting and computing the fast Fourier transform (FFT), one can obtain the savings in processor count *with no time loss.* We now describe the algorithmic setting, which generalizes an analogous discussion in [31].

Say that we are given an algorithm A that runs in synchronous mode on a parallel architecture whose interprocessor communication structure is given by a digraph $\mathcal{G}$. Say further that Algorithm A runs on $\mathcal{G}$ in the following format:

> There is a partition of the set of vertices of $\mathcal{G}$ (which are the processors of the array) into sets $V_1, V_2, \cdots, V_\ell$, such that, at each moment of time, the active set of processors involves at most one processor from each set V_i.

We call each set V_i a *block of the graph* $\mathcal{G}$, and we call Algorithm A an (ℓ-)*block-structured algorithm.* Consider now the following modification of the scenario just described.

- Label the vertices/processors of $\mathcal{G}$ in any way that assigns a different label to each processor in each block V_i; clearly $N =_{\text{def}} \max_i |V_i|$ labels suffice.
- Construct the ℓ-vertex graph/processor array $\mathcal{G}'$ that has a vertex v_i for each block V_i in the partition of $\mathcal{G}$, and that has an arc from vertex v_a to vertex v_b just when, in $\mathcal{G}$, some vertex in block V_a has an arc to some vertex in block V_b. Give each vertex v_i of $\mathcal{G}'$ the capability to simulate each processor in the block V_i of $\mathcal{G}$. (This is easy with arrays of identical processors.)
- Modify Algorithm A to obtain Algorithm A', which operates as follows. Each message generated by Algorithm A' is a message generated by Algorithm A, augmented with the label (address) of the processor of $\mathcal{G}$ that the message is intended for.
- Run Algorithm A' on graph $\mathcal{G}'$ as follows.
  - If an initial message M of Algorithm A would go to processor v in block V_i of $\mathcal{G}$, then Algorithm A' tags message M with the label of v and sends it to processor v_i of $\mathcal{G}'$.
  - When a processor of $\mathcal{G}'$ completes a task of Algorithm A', it tags each message M that it has generated with the label of the processor v (in block V_i) of $\mathcal{G}$ that Algorithm A would send the message to on $\mathcal{G}$. It then sends the message to processor v_i of $\mathcal{G}'$.
- A processor v_i of $\mathcal{G}'$ uses the label attached to incoming messages to determine which processor in block V_i of $\mathcal{G}$ to simulate during a given step of Algorithm A'. The block-oriented character of Algorithm A guarantees that a processor of $\mathcal{G}'$ is never asked to simulate more than one processor of $\mathcal{G}$ at a time.

It is transparent that Algorithm A' is functionally equivalent to Algorithm A; moreover, the only overhead for running the former algorithm to simulate the latter resides in the process of appending, sending, and decoding the processor-labels, which can be assumed to be bit-strings of length at most $\log N$. This overhead allows us to simulate a large processor array with an ℓ-processor array.

When $\mathcal{G}$ is a Cayley graph $\mathbf{Cay}(\Pi)$, and $\mathcal{G}'$ is a coset graph $\mathbf{Cos}(\mathbf{Gr}(\Pi); H; \Pi)$ of $\mathcal{G}$ by the subgroup H, then the scenario just described is often simplified somewhat, for each block V_i of $\mathcal{G}$ has the same number of processors, $|H|$. Thus, the effect of Algorithm A on array $\mathcal{G}$ is obtained on a processor array of size $|\mathcal{G}|/|H|$. In this case, we say that Algorithm A is *H-blocked.*

In certain instances, we can do even better. Consider, for illustration, executing the FFT algorithm on the butterfly network (a Cayley graph whose structure matches

the data dependencies of the algorithm; cf. §4.1.B and [1, Chap. 7]. This algorithm runs on the network level by level (using the natural leveling of the network): At each time t, each processor at level t of the network takes in inputs x_1 and x_2 on its two input ports, using the source input ports to distinguish x_1 from x_2; it computes two linear functions $L_{1,t}(x_1, x_2)$ and $L_{2,t}(x_1, x_2)$ of the inputs; it sends $L_{1,t}$ out on its first output port and $L_{2,t}$ out on its second output port. Thus, at any specific moment, only one level of processors in the network is active. One sees that this computation is so carefully choreographed that no addressing mechanism is needed to determine what a processor is to do: A processor can determine what role to play — i.e., for which t to compute $L_{1,t}$ and $L_{2,t}$ — merely by *keeping track of the time.* Generalizing from this example, we call an H-blocked algorithm for the Cayley graph $\mathcal{G}$ *orchestrated* if one can label each processor in each block of $\mathcal{G}$ (i.e., coset of H) with a set of time-stamps that indicate the times when that processor is active while executing the algorithm, *independently of the input data.* As with our example, one can execute an orchestrated blocked algorithm on the coset graph using only a clock (either global or one per processor) to tell each processor of the coset graph when it is to play which role.

In the very special case of our example of the FFT algorithm on the butterfly network, we encounter the potential for even further simplification: In the FFT algorithm, the differences among the various linear functions $L_{i,t}$, as t varies, reside in a parameter ω_t that enters into the computation of $L_{i,t}$. Since each $\omega_t = \omega_{t-1}^2$, a further algorithmic simplification is possible: If we have each processor square its current parameter before computing its linear functions, then we shall have just two fixed linear functions $L_i(x_1, x_2; \omega_{current})$, $i = 1, 2$, that are computed by every processor in the butterfly. In this case, there is no need for a processor of a coset graph to maintain any information about its "identity": *Every processor, in every block/coset, performs the same computation at every step as does every other processor.* In the case of such *oblivious* algorithms, therefore, we do not need any global or local clocks, and we do not need to devote any time to processing addresses or time-stamps: We save a large factor in hardware *at no extra cost in computation time.*[15]

We close this section with a formal summary of the preceding discussion.

THEOREM 3.6. *Let* $\mathbf{Cay}(\Pi)$ *be a Cayley graph, let* H *be a subgroup of* $\mathbf{Gr}(\Pi)$, *and let* A *be an* H*-blocked algorithm for* $\mathbf{Cay}(\Pi)$.

(a) *Algorithm* A *can be run on the coset graph* $\mathcal{G} = \mathbf{Cos}(\mathbf{Gr}(\Pi); H; \Pi)$, *slowed down by the factor* $O(\log |H|)$.

(b) *If Algorithm* A *is orchestrated, then* t *steps of the Algorithm can be run on the coset graph* $\mathcal{G}$ *in* $O(t \log t)$ *steps.*

(c) *In either of the previous circumstances, if the processors of* $\mathbf{Cay}(\Pi)$ *operate on "large" quantities, then the slowdown on the coset graph* $\mathcal{G}$ *is only* $O(1)$.

(d) *If Algorithm* A *is oblivious, then it can be run on the coset graph* $\mathcal{G}$ *with no time loss.*

As a final general remark, we note that the Cayley graph $\mathbf{Cay}(\Pi)$ can be enormous compared to its associated GAG (V, Π). Even if Π consists of two permutations, one of which cyclically permutes V and one of which switches two elements of V while holding all others fixed — so that the undirected graph underlying the GAG (V, Π) has only $|V|$ edges — the group $\mathbf{Gr}(\Pi)$ can contain $|V|!$ elements. Although we do not know of any Cayley graph-GAG-algorithm matchups that apply Theorem 3.6 to such an extreme situation, it is conceivable that such do exist.

[15] We do, however, lose the ability to pipeline computations (one per level on the butterfly).

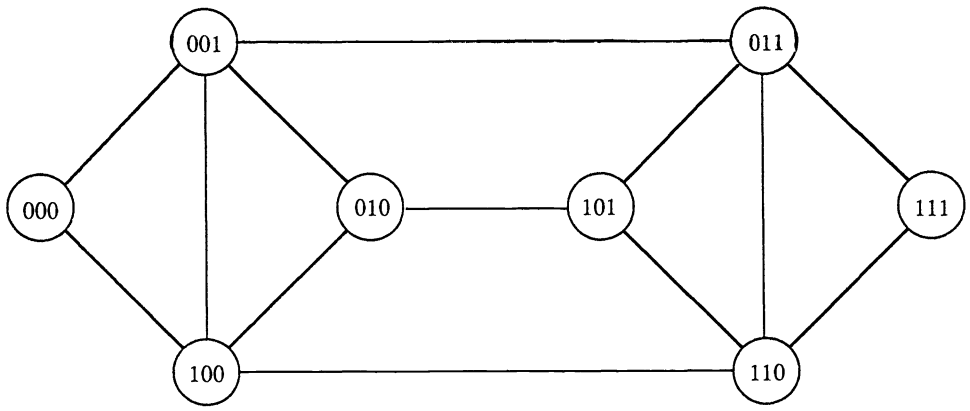


FIG. 1. *The deBruijn graph $D_2(3)$ as an undirected graph.*

4. Main results: Concrete version. In this section we demonstrate the usefulness of the development in the preceding section, by applying it to four families of graphs which are benchmarks among interconnection networks for parallel architectures:

- the *butterfly* network
- the *cube-connected cycles* network
- the *deBruijn* network
- the *shuffle-exchange* network

All four families are bounded-degree "approximations" to the hypercube network. We prove that the first two are families of Cayley graphs, both having the same wreath product of cyclic groups as underlying groups, but with slightly different generator sets; we prove that the second two are families of coset graphs of the former two families. Thus we characterize precisely and rigorously the structural similarities and differences among these families. We then discuss the implications of Theorem 3.6 for the correspondences we have exposed.

At virtually no extra cost, we establish our results for generalized versions of the four studied networks.

4.1. deBruijn and butterfly networks.

A. deBruijn graphs. Let d, n be positive integers. The *base-d order-n deBruijn graph* $\mathcal{D}_d(n)$ is the digraph whose vertices comprise the set Z_d^n, and whose arcs connect each vertex $\alpha x \in Z_d^n$, where $\alpha \in Z_d$ and $x \in Z_d^{n-1}$, to each vertex of the form $x\beta \in Z_d^n$ for some $\beta \in Z_d$; see Fig. 1 and [12], [19].

By Proposition 2.1, every deBruijn graph can be arc-labeled so as to be a GAG. One way to do this yields the correspondences we seek. For each $\beta \in Z_d$, define the permutation $\pi[\beta; d]$ of Z_d^n (n being clear from context) by

$$(\alpha x)\pi[\beta; d] = x(\alpha + \beta(\text{mod } d)),$$

for each $\alpha \in Z_d$ and $x \in Z_d^{n-1}$.[16] Label each arc of $\mathcal{D}_d(n)$ of the form $(\alpha x, x\beta)$ with the permutation $\pi[\beta - \alpha(\text{mod } d); d]$. We leave to the reader the easy verification that the described arc-labeling renders $\mathcal{D}_d(n)$ a GAG.

Let $\Pi_d^{\mathcal{D}} =_{\text{def}} \{\pi[\beta; d] \mid \beta \in Z_d\}$.

[16] For all d, the permutation $\pi[0; d]$ is just a cyclic-shift of the argument string. The permutation $\pi[0; 2]$ is termed a (*perfect*) *shuffle*, and the permutation $\pi[1; 2]$ is termed a *shuffle-exchange*.

LEMMA 4.1. *For all d,* $\mathbf{Gr}(\Pi_d^{\mathcal{D}})$ *is isomorphic to the wreath product* $Z_d \otimes_{wr} Z_n$.

Proof. Note first that for each base d and integer $\beta \in Z_d$, the permutation $\pi[\beta; d] \in \Pi_d^{\mathcal{D}}$ is equivalent to a modulo-d vector addition of $(\beta, 0, \cdots, 0)$ to the argument string/vector, followed by a one-place-left cyclic shift of the digits of the argument; $\pi[\beta; d]$ is, thus, equivalent to the permutation

$$\langle 1; \beta, 0, \cdots, 0 \rangle$$

in $Z_d \otimes_{wr} Z_n$. It follows, therefore, that $\mathbf{Gr}(\Pi_d^{\mathcal{D}})$ is a subgroup of $Z_d \otimes_{wr} Z_n$.

Next, consider the arbitrary permutation

$$\pi = \langle \alpha; \beta_0, \beta_1, \cdots, \beta_{n-1} \rangle$$

in $Z_d \otimes_{wr} Z_n$. By definition, the action of π on an element of Z_d^n consists of a modulo-d vector addition of $(\beta_0, \beta_1, \cdots, \beta_{n-1})$ to the argument, followed by α left-cyclic shifts. The action of π is, thus, equivalent to the action of the product[17]

$$\pi[\beta_0; d]\pi[\beta_1; d] \cdots \pi[\beta_{n-1}; d](\pi[0; d])^{\alpha}$$

of permutations from $\Pi_d^{\mathcal{D}}$: the first n permutations effect the vector addition, while the last α effect the cyclic shift. It follows, therefore, that $Z_d \otimes_{wr} Z_n$ is a subgroup of $\mathbf{Gr}(\Pi_d^{\mathcal{D}})$.

Lemma 4.1 follows. □

LEMMA 4.2. *For all d, n, the base-d, order-n deBruijn graph* $\mathcal{D}_d(n)$ *is isomorphic to the Coset graph*

$$\mathbf{Cos}(G; H; \Pi_d^{\mathcal{D}})$$

where $G = Z_d \otimes_{wr} Z_n$ *and* $H = \{0\} \otimes_{wr} Z_n$.

Proof. Since $\mathcal{D}_d(n)$ is a connected GAG, Lemma 4.2 will follow from Theorem 3.1, once we determine the stabilizer of Z_d^n in $Z_d \otimes_{wr} Z_n$. By Theorem 3.1, it will suffice to determine the stabilizer of the element $(0, 0, \cdots, 0) \in Z_d^n$, which is transparently $\mathbf{Gr}(\pi[0; d])$ (i.e., all possible shifts, with no additions). Using reasoning analogous to that in the proof of Lemma 4.1, one verifies that $\mathbf{Gr}(\pi[0; d])$ is isomorphic to the subgroup $\{0\} \otimes_{wr} Z_n$ of $Z_d \otimes_{wr} Z_n$. □

B. Butterfly networks. Let d, n be a positive integer. The *base-d order-n butterfly graph* $\mathcal{B}_d(n)$ has vertex-set

$$V_{n;d} = Z_n \times Z_d^n.$$

The subset $V_{n;d}^{(\ell)} = \{\ell\} \times Z_d^n$ of $V_{n;d}$ $(0 \leq \ell < n)$ is the ℓth *level* of $\mathcal{B}_d(n)$; the string $x \in Z_d^n$ is the *position-within-level* (PWL, for short) string of each vertex in $Z_n \times \{x\}$. The edges of $\mathcal{B}_d(n)$ form *d-butterflies* (i.e., copies of the complete bipartite graph $K_{d,d}$) between consecutive levels of vertices, with wraparound in the sense that level 0 is identified with level n. Each butterfly connects each vertex

$$\langle \ell,\ \beta_0\beta_1 \cdots \beta_{\ell-1}\alpha\beta_{\ell+1} \cdots \beta_{n-1} \rangle$$

on level ℓ of $\mathcal{B}_d(n)$ $(0 \leq \ell < n$; α and each β_i in $Z_d)$ with all vertices

$$\langle \ell + 1 (\mathrm{mod}\ n),\ \beta_0\beta_1 \cdots \beta_{\ell-1}\omega\beta_{\ell+1} \cdots \beta_{n-1} \rangle$$

on level $\ell + 1 (\mathrm{mod}\ n)$ of $\mathcal{B}(_d(n))$, for all $\omega \in Z_d$;[18] see Fig. 2.

[17] In the product, $(\pi[0; d])^{\alpha}$ denotes a sequence of α instances of $\pi[0; d]$.

[18] $\mathcal{B}_2(n)$ can be viewed as the FFT network with the input and output vertices (i.e., the top and bottom levels) identified.

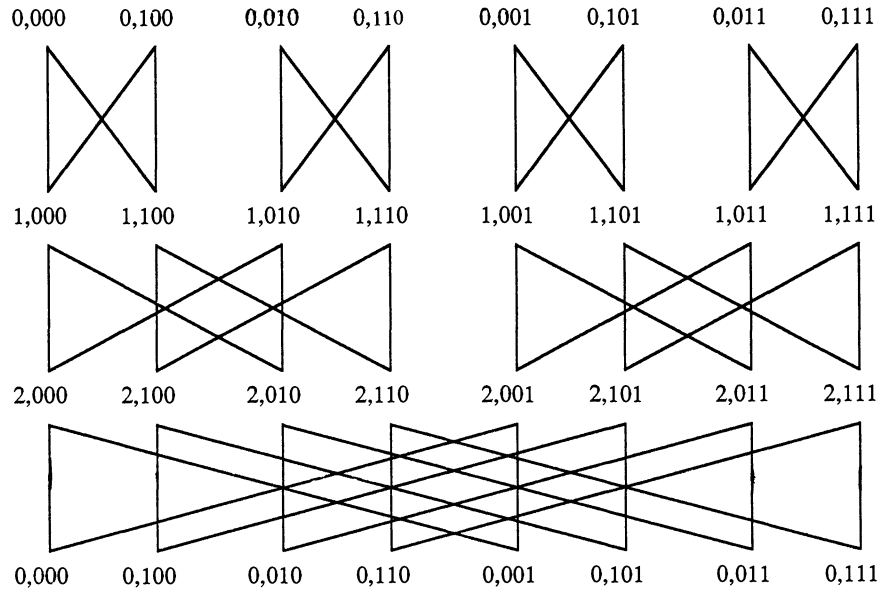


FIG. 2. *The butterfly graph* $B_2(3)$ *as an undirected graph; level* 0 *is replicated to aid visualization.*

There is a natural way to turn $\mathcal{B}_d(n)$ into a TRAG whose arcs are labeled with permutations from $\Pi_d^{\mathcal{D}}$. First, we form the *directed* version $\vec{\mathcal{B}}_d(n)$ of $\mathcal{B}_d(n)$, by orienting each edge of $\mathcal{B}_d(n)$ from level ℓ to level $\ell+1$ (before reducing $\ell+1$ modulo n). Next, we form the TRAG version $\tilde{\mathcal{B}}_d(n)$ of $\vec{\mathcal{B}}_d(n)$, as follows. For each $\ell \in Z_n$ and $\alpha, \omega \in Z_d$, we label the arc from vertex

$$\langle \ell,\ \beta_0\beta_1\cdots\beta_{\ell-1}\alpha\beta_{\ell+1}\cdots\beta_{n-1}\rangle$$

to vertex

$$\langle \ell+1(\mathrm{mod}\ n),\ \beta_0\beta_1\cdots\beta_{\ell-1}\omega\beta_{\ell+1}\cdots\beta_{n-1}\rangle$$

with the permutation $\pi[\omega-\alpha(\mathrm{mod}\ d);\ d]$.

LEMMA 4.3. *For all* d, n, *the base-*d *order-*n *butterfly network* $\tilde{\mathcal{B}}_d(n)$ *is isomorphic* (as a TRAG) *to the Cayley graph* $\mathbf{Cay}(\Pi_d^{\mathcal{D}})$.

Proof. We show that the natural correspondence between vertices of $\tilde{\mathcal{B}}_d(n)$ and vertices of $\mathbf{Cay}(\Pi_d^{\mathcal{D}})$ yields the desired isomorphism. By Lemma 4.1, the latter set of vertices comprises just the elements of $Z_d \otimes_{wr} Z_n$. Let each vertex

$$\langle \ell,\ \beta_0\beta_1\cdots\beta_{n-1}\rangle$$

($\ell \in Z_n$; each $\beta_i \in Z_d$) of $\tilde{\mathcal{B}}_d(n)$ correspond to the element

$$\langle \ell;\ \beta_0, \beta_1, \cdots, \beta_{n-1}\rangle$$

of $Z_d \otimes_{wr} Z_n$. Since this mapping is well defined and onto, it is one-to-one also. To complete the proof, we need only verify that our correspondence preserves (labeled) arcs. To simplify exposition in this verification, let us agree that all addition in the remainder of the proof is modulo d.

Every arc labeled $\pi[\gamma; d] \in \Pi_d^{\mathcal{D}}$ in $\mathbf{Cay}(\Pi_d^{\mathcal{D}})$ leads from a vertex

$$\zeta = \langle \ell;\ \beta_0, \beta_1, \cdots, \beta_{n-1} \rangle$$

to vertex $\zeta \cdot \pi[\gamma; d]$. To determine the image of this latter vertex in $\tilde{\mathcal{B}}_d(n)$, we must consider the action of $\zeta \cdot \pi[\gamma; d]$ on an arbitrary vector $(\delta_0, \delta_1, \cdots, \delta_{n-1})$, each $\delta_i \in Z_d$. By definition, this action consists of

- addition of the vector $(\beta_0, \beta_1, \cdots, \beta_{n-1})$, followed by a sequence of ℓ left-cyclic shifts,

which is the action of ζ, followed by

- addition of the vector $(\gamma, 0, 0, \cdots, 0)$, followed by a single left-cyclic shift,

which is the action of $\pi[\gamma; d]$. This is easily seen to be identical to the action of the permutation

$$\langle \ell + 1;\ \beta_0, \beta_1, \cdots, \beta_{\ell-1}, \beta_\ell + \gamma, \beta_{\ell+1}, \cdots, \beta_{n-1} \rangle$$

of $Z_d \otimes_{wr} Z_n$. There is, thus, an arc labeled $\pi[\gamma; d]$ in $\tilde{\mathcal{B}}_d(n)$, from the vertex corresponding to ζ to the vertex corresponding to $\zeta \cdot \pi[\gamma; d]$.

Every arc labeled $\pi[\gamma; d]$ in $\tilde{\mathcal{B}}_d(n)$ leads from a vertex

$$v = \langle \ell,\ \beta_0 \beta_1 \cdots \beta_{\ell-1} \beta_\ell \beta_{\ell+1} \cdots \beta_{n-1} \rangle$$

to vertex

$$\langle \ell + 1,\ \beta_0 \beta_1 \cdots \beta_{\ell-1} (\beta_\ell + \gamma) \beta_{\ell+1} \cdots \beta_{n-1} \rangle.$$

By our correspondence, this latter vertex corresponds to the element

$$\eta = \langle \ell + 1;\ \beta_0, \beta_1, \cdots, \beta_{\ell-1}, (\beta_\ell + \gamma), \beta_{\ell+1}, \cdots, \beta_{n-1} \rangle$$

of $Z_d \otimes_{wr} Z_n$. If we let ζ denote the element of $Z_d \otimes_{wr} Z_n$ corresponding to vertex v, then reasoning similar to that in the previous paragraph verifies that $\eta = \zeta \cdot \pi[\gamma; d]$. This verifies that labeled arcs between vertices in $\tilde{\mathcal{B}}_d(n)$ betoken like-labeled arcs between the corresponding vertices of $\mathbf{Cay}(\Pi)$. □

Lemmas 4.1, 4.2, and 4.3 establish our first concrete correspondence; see Fig. 3.

THEOREM 4.4. *For all d, n, the base-d order-n deBruijn graph $\mathcal{D}_d(n)$ is a coset graph of the base-d order-n butterfly network $\tilde{\mathcal{B}}_d(n)$.*

Our development to this point allows us to infer, using Corollary 3.3, that butterfly networks contain a lot of disjoint large complete trees, thus yielding a simple algebraic proof of a somewhat complicated combinatorial result from [6].

COROLLARY 4.5. *The base-d order-n butterfly network $\tilde{\mathcal{B}}_d(n)$ contains n disjoint copies of the height-$(n-1)$ complete d-ary tree.*

Proof. It is trivial to verify that the deBruijn graph $\mathcal{D}_d(n)$ contains the height-$(n-1)$ complete d-ary tree as a subgraph. The result, therefore, follows directly from Theorem 4.4, Lemma 4.7, and Corollary 3.3. □

C. Algorithmic consequences. The advertised algorithmic consequences of Theorem 3.6 follow from Theorem 4.1 coupled with an analysis of the structure of the cosets of the group[19] $H_0 = \{0\} \otimes_{wr} Z_n$ in the group $Z_d \otimes_{wr} Z_n$. One can prove without much difficulty (cf. the verification of the Claim in the proof of Lemma 5.1) that each such coset contains (under the correspondence of Lemma 4.3) precisely one vertex

[19] By Lemma 4.1, this is the subgroup that yields the structure of the deBruijn graph.

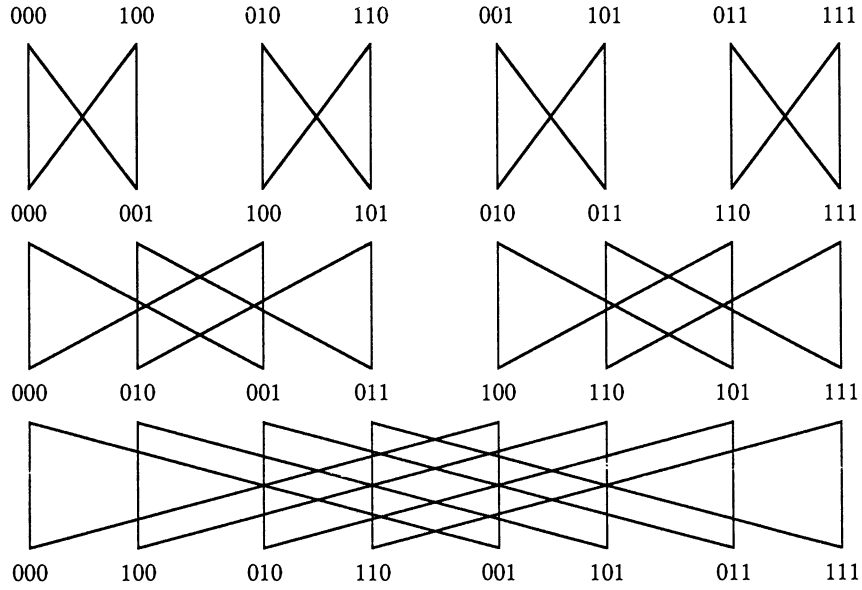


FIG. 3. $B_2(3)$ *with the shuffle-oriented labeling that exposes* $D_2(3)$ *as a coset graph; level* 0 *is replicated to aid visualization.*

from each level of $\tilde{\mathcal{B}}_d(n)$. It follows that the H_0-block-structured algorithms include all algorithms that proceed through $\tilde{\mathcal{B}}_d(n)$ level by level. This class of algorithms includes the so-called *normal* algorithms of [31] and the *ascend-descend* algorithms of [22]. Thus, any algorithm in this class can be executed on the order-n deBruijn graph essentially as fast as it can on the (much larger) order-n butterfly graph. (With a bit of effort, one can replace the word "essentially" in the previous sentence by the word "exactly"; cf. [25].) The reader can readily supply details.

4.2. Shuffle-exchange and CCC networks.

A. Shuffle-exchange graphs. Let d, n be positive integers. The *base-d order-n shuffle-exchange graph* $\mathcal{S}_d(n)$ is the GAG whose vertices comprise the set Z_d^n, and whose (labeled) arcs are specified by the permutations $\pi(\beta; d)$ of Z_d^n ($\beta \in Z_d$ and n being clear from context) defined as follows. For each $\alpha \in Z_d$ and $x \in Z_d^{n-1}$,

$$(\alpha x)\pi(0; d) = (\alpha x)\pi[0; d] = x\alpha$$

and, for $\beta \neq 0$,

$$(x\alpha)\pi(\beta; d) = x(\alpha + \beta(\text{mod } d));$$

see Fig. 4.[20] Let $\Pi_d^{\mathcal{S}} =_{\text{def}} \{\pi(\beta; d) \mid \beta \in Z_d\}$.

LEMMA 4.6. *For all d,* $\mathbf{Gr}(\Pi_d^{\mathcal{S}})$ *is isomorphic to the wreath product* $Z_d \otimes_{wr} Z_n$.[21]

Proof. By Lemma 4.1, it will suffice to prove that for all d, $\mathbf{Gr}(\Pi_d^{\mathcal{S}}) = \mathbf{Gr}(\Pi_d^{\mathcal{D}})$. To this end, note that for all d: $\pi[0; d] = \pi(0; d)$, and for $\beta \in Z_d - \{0\}$,

$$\pi[\beta; d] = \pi(\beta; d)\pi(0; d);$$

[20] The permutation $\pi(1; 2)$ is termed an *exchange.*

[21] The set $\Pi_d^{\mathcal{S}}$ is often called the *standard* set of generators for $Z_d \otimes_{wr} Z_n$.

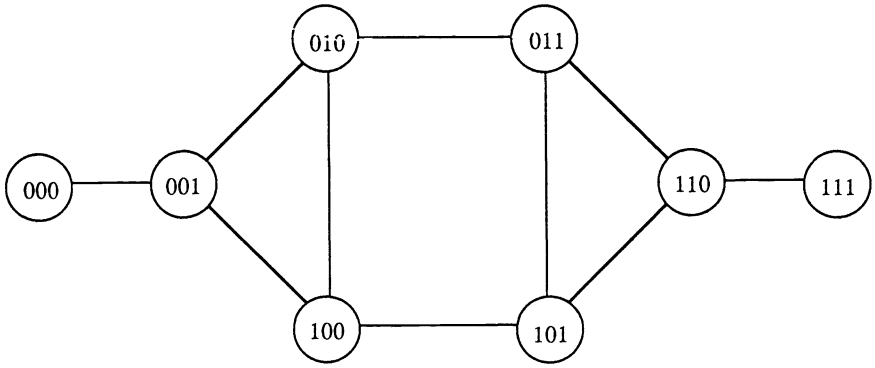


FIG. 4. *The shuffle-exchange graph $S_2(3)$ as an undirected graph.*

and

$$\pi(\beta; d) = \pi[\beta; d](\pi[0; d])^{n-1}. \qquad \square$$

LEMMA 4.7. *For all d, n, the base-d order-n shuffle-exchange graph $\mathcal{S}_d(n)$ is isomorphic to the Coset graph*

$$\mathbf{Cos}(G; H; \Pi_d^{\mathcal{S}})$$

where $G = Z_d \otimes_{wr} Z_n$ and $H = \{0\} \otimes_{wr} Z_n$.

Proof. The proof is virtually identical to that of Lemma 4.2, so is left to the reader. $\square$

B. Cube-connected cycles networks. Let d, n be positive integers. The *base-d order-n cube-connected cycles graph* (CCC*graph*, for short) $\mathcal{C}_d(n)$ has vertex-set

$$V_{n;d} = Z_n \times Z_d^n.$$

The subset $V_{n;d}^{(\ell)} = \{\ell\} \times Z_d^n$ of $V_{n;d}$ $(0 \le \ell < n)$ is the ℓth *level* of $\mathcal{C}_d(n)$. The edges of $\mathcal{C}_d(n)$ are of two varieties. First there are the *interlevel* edges that connect each vertex

$$\langle \ell,\ \beta_0\beta_1, \cdots, \beta_{n-1} \rangle$$

on level ℓ of $\mathcal{C}_d(n)$ $(0 \le \ell < n$; α and each $\beta_i \in Z_d)$ with the corresponding vertex

$$\langle \ell + 1 (\mathrm{mod}\ n),\ \beta_0\beta_1 \cdots \beta_{n-1} \rangle$$

on level $\ell + 1 (\mathrm{mod}\ n)$ of $\mathcal{C}_d(n)$. The remaining, *intralevel,* edges form *d-cliques* (i.e., copies of the complete graph K_d), as follows: On each level $0 \le \ell < n$, each vertex

$$\langle \ell,\ \beta_0\beta_1 \cdots \beta_{\ell-1}\alpha\beta_{\ell+1} \cdots \beta_{n-1} \rangle$$

(α and each β_i in Z_d) is adjacent to all vertices

$$\langle \ell,\ \beta_0\beta_1 \cdots \beta_{\ell-1}\omega\beta_{\ell+1} \cdots \beta_{n-1} \rangle$$

for all $\omega \in Z_d$.

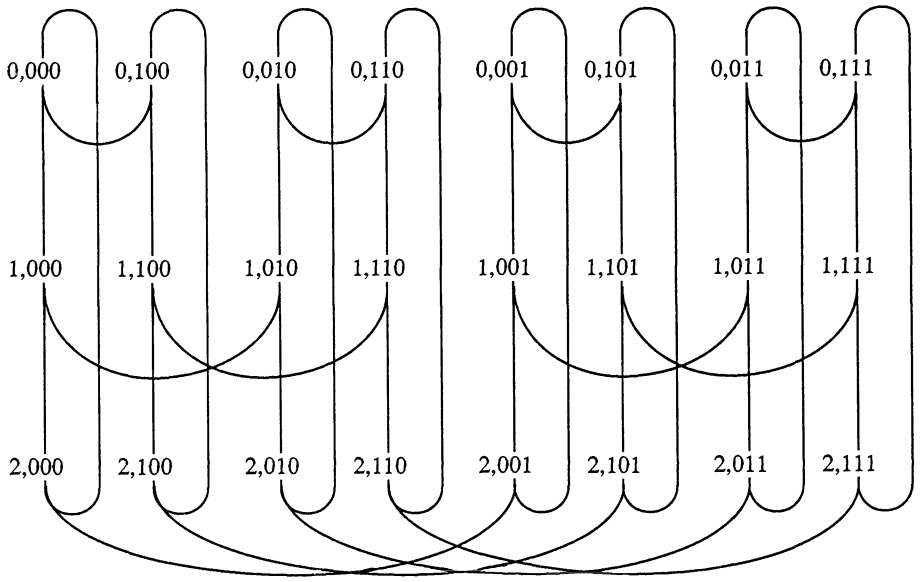


FIG. 5. *The* CCC *graph* $C_2(3)$ *as an undirected graph.*

There is a natural way to turn $\mathcal{C}_d(n)$ into a TRAG whose arcs are labeled with permutations from Π_d^S. First, we form the *directed* version $\vec{\mathcal{C}}_d(n)$ of $\mathcal{C}_d(n)$, by orienting each interlevel edge of $\mathcal{C}_d(n)$ from level ℓ to level $\ell+1$ (before reducing $\ell+1$ modulo n); then, we replace each intralevel edge of $\mathcal{C}_d(n)$ by mated opposing arcs. Next, we form the TRAG version $\tilde{\mathcal{C}}_d(n)$ of $\vec{\mathcal{C}}_d(n)$, as follows. First, we label each interlevel arc with the permutation $\pi(0;d)$; then, for each $\ell \in Z_n$ and $\alpha, \omega \in Z_d$, we label the arc from vertex

$$\langle \ell,\ \beta_0\beta_1\cdots\beta_{\ell-1}\alpha\beta_{\ell+1}\cdots\beta_{n-1}\rangle$$

to vertex

$$\langle \ell,\ \beta_0\beta_1\cdots\beta_{\ell-1}\omega\beta_{\ell+1}\cdots\beta_{n-1}\rangle$$

with the permutation $\pi(\omega-\alpha(\mathrm{mod}\ d);\ d)$; see Fig. 5.

LEMMA 4.8. *For all* d, n, *the base-*d *order-*n CCC *network* $\tilde{\mathcal{C}}_d(n)$ *is isomorphic* (as a TRAG) *to the Cayley graph* $\mathbf{Cay}(\Pi_d^S)$.

THEOREM 4.9. *For all* d, n, *the base-*d *order-*n *shuffle-exchange graph* $\mathcal{S}_d(n)$ *is a coset graph of the base-*d *order-*n CCC *network* $\tilde{\mathcal{C}}_d(n)$.

Proof sketch. The proofs of Lemma 4.8 and Theorem 4.9 are almost identical to those of Lemma 4.3 and Theorem 4.4, respectively, so are left to the reader; see Fig. 6. □

Part II: Graph-theoretic development.

5. Simulating butterflies by shuffles. We now shift gears to a purely graph-theoretic framework, in which we develop a new, efficient simulation of butterfly-oriented graphs by like-sized shuffle-oriented graphs. The major insight that yields this simulation (which resides in Lemma 5.2) derives from the development in Part I. Informally, one can view the relevant insight as follows. It is well known that if one "collapses" the order-n butterfly graph $\mathcal{B}_2(n)$ by contracting all vertices having

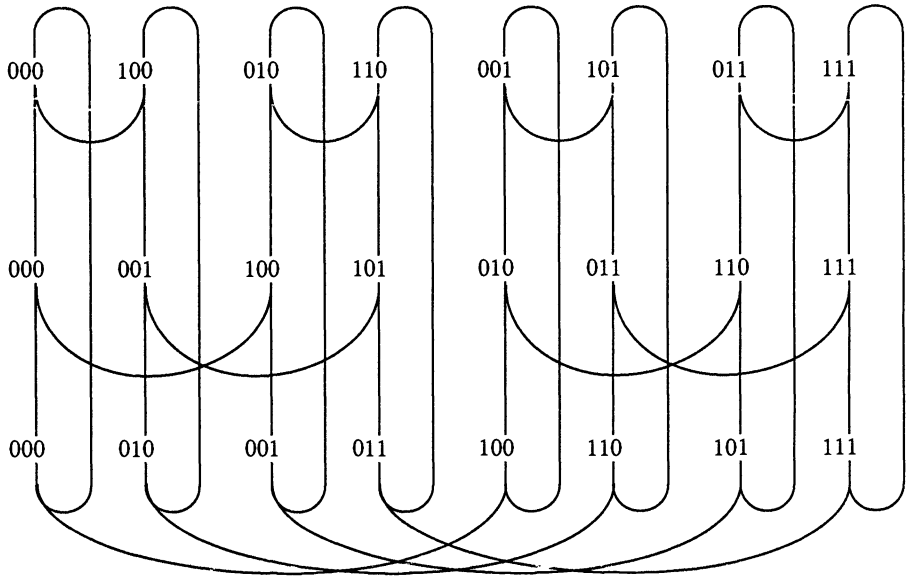


FIG. 6. $C_2(3)$ *with the shuffle-oriented labeling that exposes* $S_2(3)$ *as a coset graph.*

the same PWL string (cf. §4.1B) into a single vertex, then the resulting graph is (isomorphic to) the n-dimensional hypercube. The proof of Theorem 4.4 affords us the following new analogue of this fact. Let us label each vertex of $\mathcal{B}_2(n)$ (which, by Lemma 4.3, is an element of the group $Z_2 \otimes_{wr} Z_n$) with the name of the element of $(Z_2 \otimes_{wr} Z_n)/(\{0\} \otimes_{wr} Z_n)$ to which it belongs. (By Lemma 4.2, these cosets are the vertices of the associated deBruijn graph $\mathcal{D}_2(n)$.) If one now "collapses" $\mathcal{B}_2(n)$ by contracting all vertices having the same coset-label into a single vertex, then the resulting graph is (isomorphic to) the order-n deBruijn graph $\mathcal{D}_2(n)$.

We present here only a simulation of the butterfly graph $\mathcal{B}_2(n)$ by the deBruijn graph $\mathcal{D}_2(n + \lceil \log_2 n \rceil)$, since focusing on this pair of graphs simplifies exposition and allows us to minimize constant factors. At the cost of some clerical complication, we could focus on graphs of any base d; indeed, the bases of the simulated and simulating graphs need not even be the same, although constant factors must be adjusted if the bases are allowed to differ. At the cost of small constant factors, we could replace the butterfly graph $\mathcal{B}_2(n)$ by the CCC graph $\mathcal{C}_2(n)$ and/or replace the deBruijn graph $\mathcal{D}_2(n + \lceil \log_2 n \rceil)$ by the shuffle-exchange graph $\mathcal{S}_2(n + \lceil \log_2 n \rceil)$. We leave these fine points to the reader.

Certain conventions will simplify our exposition.

- We elide the subscript "2" from the names of our graphs, since we consider only the base-2 version henceforth.
- We let $\Lambda(n) =_{\text{def}} \lceil \log_2 n \rceil$.
- We term an edge of $\mathcal{B}(n)$ a *straight-edge* if it leaves the PWL string unchanged, i.e., has the form $(\langle \ell, x \rangle, \langle \ell + 1 (\text{mod } n), x \rangle)$; we term the edge a *cross-edge* if it changes the PWL string.

While proving our main simulation result, we set off a series of fundamental lemmas that are likely to be useful in other contexts also.

THEOREM 5.1. *For all n, one can embed the order-n butterfly graph $\mathcal{B}(n)$ in the order-$(n + \Lambda(n))$ deBruijn graph $\mathcal{D}(n + \Lambda(n))$, with dilation $O(\Lambda(n))$, congestion*

$O(\Lambda(n))$, *and dynamic congestion* $O(1)$.

Proof. We develop our embedding in three steps.

1. We embed $\mathcal{B}(n)$ in the product graph $\mathcal{R}(n) \times \mathcal{D}(n)$,[22] with dilation 2 (Lemma 5.2).
2. We embed $\mathcal{R}(n) \times \mathcal{D}(n)$ in $\mathcal{D}(\Lambda(n)) \times \mathcal{D}(n)$ with unit dilation, i.e., as a subgraph (Lemma 5.3).
3. We embed $\mathcal{D}(\Lambda(n)) \times \mathcal{D}(n)$ in $\mathcal{D}(n + \Lambda(n))$, with dilation $2\Lambda(n) + 1$, via an embedding that has small dynamic congestion (Lemma 5.4).

In the course of the third embedding, we prove the amusing fact that n disjoint copies of $\mathcal{D}(n)$ can be embedded simultaneously in $\mathcal{D}(n + \Lambda(n))$, with dilation $2\Lambda(n) + 1$.

LEMMA 5.2. *For all* n, *one can embed the order-*n *butterfly graph* $\mathcal{B}(n)$ *in* $\mathcal{R}(n) \times \mathcal{D}(n)$ *with dilation* 2, *congestion* 2, *and dynamic congestion* 2.

Proof. We begin by labeling the vertices of $\mathcal{B}(n)$ with strings from Z_2^n, using the *shuffle-oriented* labeling, which is implicit in the proof of Theorem 3.2; see Fig. 3. The labeling rules are as follows:

1. Label vertex $\langle 0, \vec{0} \rangle$ of $\mathcal{B}(n)$ with the string $\vec{0}$.
2. If level-ℓ vertex v ($\ell \in Z_n$) is labeled with string $L(v)$, then
   - label the *straight-edge* neighbor of vertex v on level $\ell + 1 \pmod n$ with the *shuffle* of $L(v)$.
   - label the *cross-edge* neighbor of vertex v on level $\ell + 1 \pmod n$ with the *shuffle-exchange* of $L(v)$.

CLAIM. *Each level-*ℓ *vertex of* $\mathcal{B}(n)$ *is assigned a unique label.*

Verification. Let us "unwrap" $\mathcal{B}(n)$ by making two copies of every vertex at level 0; one copy remains at level 0, in that it is connected to vertices at level 1, while one copy participates in a new level n, in that it is connected to vertices at level $n - 1$. Let us call the resulting graph $\mathcal{B}'(n)$. (As noted in footnote 18, $\mathcal{B}'(n)$ is the 2^n-input FFT graph.) There is a unique "downward" path in $\mathcal{B}'(n)$ from vertex $\langle 0, \vec{0} \rangle$ to each vertex on level n, i.e., a path that increases level-number at each step: to wit, every level-0 vertex of $\mathcal{B}'(n)$ is the root of a complete binary tree whose leaves are all of the level-n vertices. Moreover, the unique path from vertex $\langle 0, \vec{0} \rangle$ to each level-n vertex is recorded in the label of the vertex: the kth edge of the path is a straight-edge (respectively, a cross-edge) just when the kth symbol of the label is a 0 (respectively, a 1). It follows that the labels of all level-n vertices are distinct. But, the same must now be true of every level of $\mathcal{B}'(n)$ since, according to rule 2, the label of an arbitrary vertex u at level ℓ is just the $(n - \ell)$-fold shuffle of the label of the level-n vertex reached by following $n - \ell$ straight-edges from u. Since the vertex-labels in $\mathcal{B}'(n)$ are inherited from those in $\mathcal{B}(n)$, the claimed uniqueness follows. □

Now, isolate any two consecutive levels of the labeled $\mathcal{B}(n)$, together with the 2^{n+1} edges that connect the levels; cf. Fig. 7. Produce a 2^n-vertex graph $\mathcal{G}_n$ from the isolated levels by identifying like-labeled vertices and removing the self-loops from the vertices labeled $\vec{0}$ and $\vec{1}$. Our remarks about how labels of adjacent vertices are either the shuffle or the shuffle-exchange of each other renders transparent the following claim, which exposes the contraction-mapping discussed at the beginning of the section.

CLAIM. *For all choices of adjacent levels of* $\mathcal{B}(n)$, *the graph* $\mathcal{G}_n$ *is isomorphic to* $\mathcal{D}(n)$.

The lemma is now direct: To embed $\mathcal{B}(n)$ in $\mathcal{R}(n) \times \mathcal{D}(n)$:

[22] $\mathcal{R}(n)$ is the n-vertex *cycle*, i.e., the graph whose vertices comprise the set Z_n and whose edges connect vertices i and $i + 1 \bmod n$.

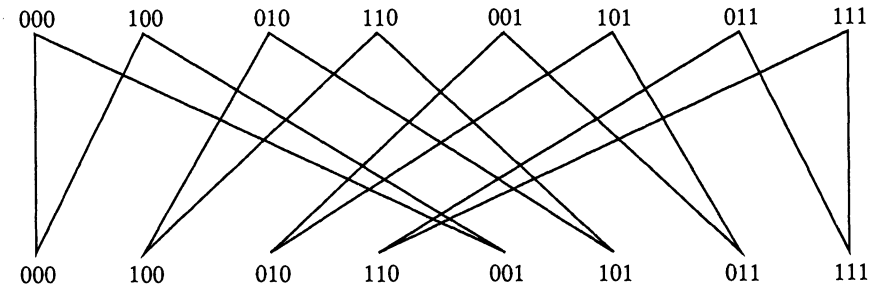


FIG. 7. *Two consecutive levels of $B_2(3)$ with the shuffle-oriented labeling; "columns" are permuted to help visualize the identification.*

- Label the vertices of $\mathcal{B}(n)$ as indicated. Assign the level-ℓ vertex of $\mathcal{B}(n)$ that gets label x to vertex x of copy ℓ of $\mathcal{D}(n)$.
- Consider a straight-edge (respectively, a cross-edge) e of $\mathcal{B}(n)$ that connects the vertex labeled x on level ℓ with the vertex labeled y on level $\ell + 1 \pmod n$. Route edge e within $\mathcal{R}(n) \times \mathcal{D}(n)$ via the length-2 path consisting of
  - the shuffle edge (respectively, the shuffle-exchange edge) that connects vertex x and vertex y in copy ℓ of $\mathcal{D}(n)$,
  - the edge that connects the instances of vertex y in copies ℓ and $\ell + 1 \pmod n$ of $\mathcal{D}(n)$.

The validity and efficiency of the embedding should be obvious. □

We state and prove Lemmas 5.3 and 5.5 in a somewhat more general context than is needed for Theorem 5.1.

LEMMA 5.3. *For all m and n, one can embed $\mathcal{R}(m) \times \mathcal{D}(n)$ in $\mathcal{D}(\Lambda(m)) \times \mathcal{D}(n)$ with unit dilation, i.e., as a subgraph.*

Proof. It is proved in [32] that the deBruijn graph $\mathcal{D}(n)$ is *pancyclic*, i.e., that it contains every cycle $\mathcal{R}(k)$, for $1 \leq k \leq 2^n$, as a subgraph. (In fact, $\mathcal{D}(n)$ is pancyclic even as a directed graph [32]; and this property persists even for arbitrary base-d deBruijn graphs $\mathcal{D}_d(n)$ [19].) This pancyclicity guarantees that $\mathcal{D}(\Lambda(m))$ contains $\mathcal{R}(m)$ as a subgraph. Lemma 5.2 follows. □

LEMMA 5.4. *For all m and n, one can embed $\mathcal{D}(m) \times \mathcal{D}(n)$ in the order-$(m+n)$ deBruijn graph $\mathcal{D}(m+n)$, with dilation $2m+1$, congestion $8m+2$, and dynamic congestion 4.*

Proof. We build on the following amusing fact.

LEMMA 5.5. *For all m and n, one can embed 2^m vertex-disjoint copies of $\mathcal{D}(n)$ in $\mathcal{D}(m+n)$ with dilation $2m+1$, and congestion at most $4m+2$.*

Proof. Let us be given 2^m copies of $\mathcal{D}(n)$. Label each with a distinct string $x \in Z_2^m$, so we can talk about copy x of $\mathcal{D}(n)$, which we denote $\mathcal{D}_x$.

For each x, we embed $\mathcal{D}_x$ in $\mathcal{D}(m+n)$ as follows. We assign vertex v of $\mathcal{D}_x$ to vertex xv of $\mathcal{D}(m+n)$. We route edge $(\beta v, v\gamma)$ of $\mathcal{D}_x$ $(\beta, \gamma \in Z_2;\ \gamma \in \{\beta, 1-\beta\};\ v \in Z_2^{n-1})$ via the following length-$(2m+1)$ path in $\mathcal{D}(m+n)$ that connects vertices $x\beta v$ and $xv\gamma$: Let $x = \xi_1 \xi_2 \cdots \xi_m$.

$$
\begin{array}{rcll}
x\beta v & = & \xi_1\xi_2\xi_3\cdots\xi_m\beta v & \\
 & \leftrightarrow & \xi_2\xi_3\cdots\xi_m\beta v\gamma & \text{shuffle/shuffle-exchange} \\
 & \leftrightarrow & \xi_3\cdots\xi_m\beta v\gamma\xi_1 & \text{shuffle/shuffle-exchange} \\
 & \cdots & & \\
 & \leftrightarrow & v\gamma\xi_1\xi_2\xi_3\cdots\xi_{m-1}\xi_m & \text{shuffle/shuffle-exchange} \\
 & \leftrightarrow & \xi_m v\gamma\xi_1\xi_2\xi_3\cdots\xi_{m-1} & \text{shuffle} \\
 & \cdots & & \\
 & \leftrightarrow & \xi_2\xi_3\cdots\xi_m v\gamma\xi_1 & \text{shuffle} \\
 & \leftrightarrow & \xi_1\xi_2\xi_3\cdots\xi_m v\gamma & \text{shuffle} \\
 & = & xv\gamma. &
\end{array}
$$

In the first $m+1$ of these edges, the choice between the shuffle and the shuffle-exchange edges of $\mathcal{D}(m+n)$ is dictated by whether the symbol being rotated to the end of the string needs to be complemented (calling for the shuffle-exchange edge) or not (so the shuffle edge is needed).

The claimed dilation of the embedding is clear from our description of the routing procedure. Note that some of the routing paths are longer than necessary. (For instance, a length-1 path will suffice when x, β, and v are all 0's, and $\gamma = 1$.) This extra length does not affect dilation, which is a worst-case measure; and, it will allow us to *decrease* the dynamic congestion of the final embedding that builds on this one.

To see that the congestion of the embedding is at most $4m + 2$, say that we are told that an edge of $\mathcal{D}(m+n)$ is being used as the kth edge in a routing path. Then, if $k > m + 1$, that information completely specifies x, v, and γ; if $k \leq m + 1$, that information completely specifies v, β, and γ, and all but one bit of x. It follows that each edge of $\mathcal{D}(m+n)$ can be used to route at most two edges in each of the $2m + 1$ steps of the routing. The bound follows.

We remark in passing that the dilation of the embedding increases to $3m+2$ when one substitutes the shuffle-exchange graph for the deBruijn graph. □

Proof of Lemma 5.3. We now show how to extend the embedding of Lemma 5.5 to yield the desired embedding of $\mathcal{D}(m) \times \mathcal{D}(n)$ in $\mathcal{D}(m+n)$.

The assignment portion of the embedding is straightforward: We assign vertex $\langle x, v\rangle$ of $\mathcal{D}(m)\times\mathcal{D}(n)$ ($x \in Z_2^m$, $v \in Z_2^n$) to vertex xv of $\mathcal{D}(m+n)$. Thus, the subgraph $\{x\} \times \mathcal{D}(n)$ of $\mathcal{D}(m) \times \mathcal{D}(n)$ plays the role of $\mathcal{D}_x$. This assignment allows us to invoke the embedding of Lemma 5.5 to route the edges of $\mathcal{D}(m) \times \mathcal{D}(n)$ that connect vertices in different copies of $\mathcal{D}(m)$, i.e., that derive from edges of $\mathcal{D}(n)$. To complete the embedding, we need, therefore, specify only how to route the edges of $\mathcal{D}(m) \times \mathcal{D}(n)$ that connect vertices in different copies of $\mathcal{D}(n)$, i.e., that derive from edges of $\mathcal{D}(m)$.

We accomplish this second routing task by a technique that is almost identical to the technique of Lemma 5.5. Every edge that we must route in this second task connects a vertex u in some copy, $\mathcal{D}_x$, of $\mathcal{D}(n)$ *to the corresponding vertex* u in another copy, $\mathcal{D}_y$, where x and y are adjacent vertices of $\mathcal{D}(m)$. Thus, the path that realizes the edge must connect two vertices of the form βzv and $z\gamma v$ ($\beta, \gamma \in Z_2$; $\gamma \in \{\beta, 1-\beta\}$; $z \in Z_2^{m-1}$; $v \in Z_2^n$). We realize this path as follows. Let $z = \zeta_1\zeta_2\cdots\zeta_{m-1}$.

$$
\begin{array}{lcll}
\beta z v & = & \beta\zeta_1\zeta_2\cdots\zeta_{m-1}v & \\
 & \leftrightarrow & \zeta_1\zeta_2\cdots\zeta_{m-1}v\zeta_1 & \text{shuffle/shuffle-exchange} \\
 & \leftrightarrow & \zeta_2\cdots\zeta_{m-1}v\zeta_1\zeta_2 & \text{shuffle/shuffle-exchange} \\
 & \cdots & & \\
 & \leftrightarrow & \zeta_{m-1}v\zeta_1\zeta_2\cdots\zeta_{m-1} & \text{shuffle/shuffle-exchange} \\
 & \leftrightarrow & v\zeta_1\zeta_2\cdots\zeta_{m-1}\gamma & \text{shuffle/shuffle-exchange} \\
 & \leftrightarrow & \gamma v\zeta_1\zeta_2\cdots\zeta_{m-1} & \text{shuffle} \\
 & \leftrightarrow & \zeta_{m-1}\gamma v\zeta_1\zeta_2\cdots\zeta_{m-2} & \text{shuffle} \\
 & \cdots & & \\
 & \leftrightarrow & \zeta_2\cdots\zeta_{m-1}\gamma v\zeta_1 & \text{shuffle} \\
 & \leftrightarrow & \zeta_1\zeta_2\cdots\zeta_{m-1}\gamma v & \text{shuffle} \\
 & = & z\gamma v. &
\end{array}
$$

As in the embedding of Lemma 5.5, the choice between the shuffle and shuffle-exchange edges in the first m edges of the path is dictated by whether or not the symbol being rotated to the end of the string needs to be complemented.

It is clear that the presented mappings constitute an embedding of $\mathcal{D}(m) \times \mathcal{D}(n)$ in $\mathcal{D}(m+n)$, in that the assignment and routing functions are both one-to-one. It remains, therefore, only to assess the cost of the resulting embedding.

The dilation of the embedding is the maximum of the dilations of the two stages. The routing of edges *within* copies of $\mathcal{D}(n)$ (cf. Lemma 5.5) incurs dilation $2m+1$; the just-presented routing of edges *between* copies of $\mathcal{D}(n)$ incurs dilation $2m$.

The congestion of the embedding is the sum of the congestions of the two stages of the embedding. The routing of edges *within* copies of $\mathcal{D}(n)$, in Lemma 5.5, incurs congestion $4m+2$. We show now that routing edges *between* copies of $\mathcal{D}(n)$ incurs congestion $8m$. To wit, say that we are told that an edge of $\mathcal{D}(m+n)$ is being used as the kth edge in a routing path. Using the notation in our routing algorithm above: that information completely specifies the strings v and z, but it generally leaves both β and γ unspecified, it follows that each edge of $\mathcal{D}(m+n)$ can be used to route at most four edges in each of the $2m$ steps of the routing. The bound follows.

To determine the dynamic congestion of the embedding, assume that when a single step of a parallel algorithm traverses edges of $\mathcal{D}(m) \times \mathcal{D}(n)$, the associated routing paths within $\mathcal{D}(m+n)$ are traversed *in lockstep*. Under this assumption, at each step of the simulation, we know that edge e of $\mathcal{D}(m+n)$ is being used as the kth edge in a routing path, since the same is true of *all* edges of $\mathcal{D}(m+n)$ that are being used at that step. It follows that the "uncertainty" about which edge of $\mathcal{D}(m) \times \mathcal{D}(n)$ is being routed across a given edge of $\mathcal{D}(m+n)$ at a given step — which quantity bounds the dynamic congestion of the embedding from above — is less than the congestion of the embedding by the factor $2m$. □

Proof of Theorem 5.1. By composing the embeddings of Lemmas 5.2, 5.3, and 5.4, we obtain an embedding of $\mathcal{B}(n)$ in $\mathcal{D}(n+\Lambda(n))$. By the analyses of the lemmas, we see that the dilation of the composite embedding is $4\Lambda(n)+2$, its congestion is $\leq 24\Lambda(n)+4$, and its dynamic congestion is $\leq 12+o(1)$. □

Acknowledgment. It is a pleasure to thank the following people for their helpful comments and suggestions, which improved both the presentation and technical development of this paper: Dave Barrington, Sandeep Bhatt, Fan Chung, Don Coppersmith, Tuvi Etzion, Lenny Heath, Tom Leighton, Charles Leiserson, Avraham Lempel, Charles Neville, Seshaiyen Raghuram, and Adi Shamir.

REFERENCES

[1] S. B. AKERS AND B. KRISHNAMURTHY, *Group graphs as interconnection networks*, 14th IEEE International Conference on Fault-tolerant Computing, (1984), pp. 424–427.

[2] ———, *A group-theoretic model for symmetric interconnection networks parallel processing*, International Conference Parallel Processing, (1986), pp. 216–233.

[3] ———, *On group graphs and their fault tolerance*, IEEE Trans. Comput. C-36, (1987), pp. 885–888.

[4] M. BAUMSLAG AND A. L. ROSENBERG, *Cayley graphs that are almost direct products*, Univ. of Massachusetts, Amherst, MA, (In preparation), 1989.

[5] S. N. BHATT, F. R. K. CHUNG, J.-W. HONG, F. T. LEIGHTON, AND A. L. ROSENBERG, *Optimal emulations by butterfly networks*, J. ACM, (1988), to appear.

[6] S. N. BHATT, F. R. K. CHUNG, F. T. LEIGHTON, AND A. L. ROSENBERG, *Efficient embeddings of trees in hypercubes*, Tech. Report, Univ. of Massachusetts, Amherst, MA, 1988, submitted for publication; (1986) *Optimal simulations of tree machines*, Proc. 27th IEEE Symposium on Foundations of Computer Science.

[7] J. A. BONDY AND U. S. R. MURTY, *Graph Theory with Applications*, North-Holland, New York, 1976.

[8] J. D. BOVEY AND A. WILLIAMSON, *The probability of generating the symmetric group*, Bull. London Math. Soc., 10 (1978), pp. 91–96.

[9] G. CARLSSON, J. E. CRUTHIRDS, H. B. SEXTON, AND C. G. WRIGHT, *Interconnection networks based on a generalization of cube-connected cycles*, IEEE Trans. Comput., C-34 (1985), pp. 769–772.

[10] G. CARLSSON, M. FELLOWS, H. SEXTON, AND C. WRIGHT, *Group theory as an organizing principle in parallel processing*, J. Combin. Theory and Combin. Comput., 3 (1988).

[11] A. H. CLIFFORD AND G. B. PRESTON, *The Algebraic Theory of Semigroups*, II. Mathematical Surveys No. 7, Amer. Math. Soc., Providence, RI, 1967.

[12] N. G. DEBRUIJN, *A combinatorial problem*, Proc. Akademe van Weteschappen, 49, Part 2, (1946), pp. 758–764.

[13] A. M. DESPAIN AND D. A. PATTERSON, *X-tree / a tree structured multiprocessor architecture*, 5th International Symposium on Computer Architecture, (1978), pp. 144–151.

[14] V. FABER, *Global communication algorithms for hypercubes and other Cayley coset graphs*, SIAM J. Discrete Math., to appear.

[15] R. GINOSAR AND D. EGOZI, *Topological comparison of perfect shuffle and hypercube*, Typescript, The Technion, Haifa, Israel, (1987).

[16] D. S. GREENBERG, L. S. HEATH, AND A. L. ROSENBERG, *Optimal embeddings of butterfly-like graphs in the hypercube*, Math. Systems Theory, to appear.

[17] A. KOTZIG, *The decomposition of a directed graph into quadratic factors consisting of cycles*, (1969), pp. 27–29.

[18] F. T. LEIGHTON, B. MAGGS, AND S. RAO, *Universal packet routing algorithms*, Proc. 29th IEEE Symposium on Foundations of Computer Science, (1988), pp. 256–269.

[19] A. LEMPEL, *m-ary closed sequences*, J. Combin. Theory Ser. A, 10 (1971), pp. 253–258.

[20] J. M. HALL, *The Theory of Groups*, Macmillan, New York, 1959.

[21] D. K. PRADHAN AND M. R. SAMATHAM, *The deBruijn multiprocessor network: A versatile parallel processing and sorting network for VLSI*, IEEE Trans. Comput., 38 (1989), pp. 567–581.

[22] F. P. PREPARATA AND J. E. VUILLEMIN, *The cube-connected cycles: A versatile graph for parallel computation*, J. ACM, 24 (1981), pp. 300–309.

[23] A. L. ROSENBERG, *Data graphs and addressing schemes*, J. Comput. System. Sci., 5 (1971), pp. 193–238.

[24] ———, *An extrinsic characterization of addressable data graphs*, Discrete Math., 9 (1974), pp. 61–70.

[25] ———, *Shuffle-like interconnection networks*, Tech. Report 88-84, Univ. of Massachusetts, Amherst, MA, 1988.

[26] Y. SAAD AND M. H. SCHULTZ, *Topological properties of hypercubes*, IEEE Trans. Comput., 37 (1988), pp. 867–872.

[27] J. T. SCHWARTZ, *Ultracomputers*, ACM Trans. Program Language, 2 (1980), pp. 484–521.

[28] C. STANFILL, *Communications architecture in the connection machine system*, Tech. Report HA87-3, Thinking Machines Corp., Cambridge, MA, 1987.

[29] H. STONE, *Parallel processing with the perfect shuffle*, IEEE Trans. Comput., C-20 (1971), pp. 153–161.

[30] D. TZVIELI, *Minimal diameter double-ring networks* I: *Some very large infinite optimal fam-*

ilies, Louisiana State Univ., Baton Rouge, LA, 1988.
[31] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.
[32] M. YOELI, *Binary ring sequences*, Amer. Math. Monthly, 69 (1962), pp. 852–855.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 570–590, June 1990

013

EMBEDDING TREES IN A HYPERCUBE IS NP-COMPLETE*

A. WAGNER† AND D. G. CORNEIL‡

Abstract. An important family of graphs is the n-dimensional hypercube, the graph with 2^n nodes labelled $0, 1, \cdots, 2^n - 1$, and an edge joining two nodes whenever their binary representation differs in a single coordinate. The problem of deciding if a given source graph is a partial subgraph of an n-dimensional cube has recently been shown to be NP-complete. In this paper the same problem on a very restricted family of source graphs, trees, is considered. It is shown that the problem of determining for a given tree T and integer k if T is a partial subgraph of a k-dimensional cube is NP-complete.

Key words. NP-complete, trees, hypercube, multiprocessor, graph embedding

AMS(MOS) subject classifications. 68Q25, 68M10, 05C05, 68Q10

1. Introduction.

An important family of graphs attracting a great deal of attention lately is the n-dimensional Boolean cube or simply n-cube. An n-cube graph, Q_n, consists of 2^n vertices labelled $\{0, 1, \cdots, 2^n - 1\}$ and an edge joining two vertices whenever the binary representation of their labels differ in only one coordinate. The main source of interest in this family of graphs has been the recent appearance of several multiprocessor architectures, generically called hypercubes, whose interconnection is based on the n-cube. Typically, an order n hypercube has 2^n microprocessors, each with its own memory, that are interconnected by $n2^{n-1}$ point-to-point communication channels corresponding to the edges of an n-cube.

There are several properties of the hypercube that have contributed to its success. From an engineering perspective, current technology has made it both technically and economically feasible to build hypercubes with large numbers of vertices. As an interconnection network, it is scalable to thousands of vertices and the hardware costs grow logarithmically with the number of vertices. There is a simple routing algorithm and the interconnection has logarithmic diameter and high bandwidth. There are $d!$ shortest paths between vertices a distance d apart and, as shown by Foldes [9], the n-cube is the only bipartite graph with this property.

From a programming perspective, the hypercube's recursive structure is well-suited for divide-and-conquer-type algorithms. Many numerical applications take advantage of the fact that rings, two-dimensional meshes, higher-dimensional meshes, hexagons, and almost complete binary trees are all embeddable in a hypercube [16][3] [23][18][15]. However, these structures may not always capture the irregular computation and communication structure of many programs and the problem of embedding these irregular structures remains. This problem is not unique to hypercubes, but occurs in all multiprocessor architectures and is known as the mapping problem [4].

The problem of mapping a graph representing the computation and communication needs of the program onto the underlying physical interconnection of a multiprocessor so as to minimize the communication overhead and maximize the parallelism is called the mapping problem. In its simplest form (i.e., unit costs) this problem reduces to the subgraph isomorphism problem that is known to be NP-complete. In

* Received by the editors February 24, 1987; accepted for publication (in revised form) September 1, 1989. This work was supported by the Natural Sciences and Engineering Research Council of Canada.

† Department of Computer Science, University of British Columbia, Vancouver, British Columbia, Canada, V6T 1W5.

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

our case there is the added restriction of a fixed host. One well-studied problem for fixed hosts is the bandwidth minimization problem – given a graph G and integer k can G be embedded in a line so that any two vertices adjacent in G are at most distance k apart on the line. This problem was shown to be NP-complete even when G was a binary tree [11], but is solvable in polynomial time for fixed distance k [20]. In general, however, little is known about the NP-completeness status of the subgraph isomorphism problem for restricted hosts and there are few if any results comparable to that which is known about the bandwidth minimization problem [19]. In this paper we consider the subgraph isomorphism problem on a restricted host, the n-cube.

The problem of deciding if a graph is a subgraph of an n-cube is not new, variations of this problem first appeared over 20 years ago in the areas of switching circuits and coding theory. Early work by Deza [22] and Firsov [8] established some basic properties of isometric subgraphs (i.e., distance preserving) of the n-cube and in 1973, Djokovič [7] completely characterized these subgraphs. Later work by Garey and Graham [10] showed that there are infinite families of forbidden graphs and posed the question as to whether or not there exist efficient algorithms for deciding if a graph is a partial subgraph of an n-cube. The problem of characterizing the partial subgraphs of the n-cube was one of the open problems that appeared in the 1976 book by Bondy and Murty on graph theory [5].

It was not until 1984 that Afrati, Papadimitriou, and Papageorgiou [1] proved that deciding if a graph was a partial subgraph of a hypercube was NP-complete. The validity of their proof, however, has been challenged by Krumme and Venkataraman [17] who independently, using a different reduction technique, obtained the same result [6]. Although this solved the problem on general graphs there was no obvious way to extend either of these reduction techniques to obtain similar results on restricted families of source graphs, and trees in particular.

The problem of embedding trees into an n-cube has also been studied. Since all trees are embeddable in a sufficiently large n-cube the problem now is to find the smallest n-cube containing a given tree. Afrati, Papadimitriou, and Papageorgiou[1], and much earlier Havel and Lieble[14], noticed that for an arbitrary tree with N vertices the number of dimensions required can vary from $\log(N)$ to $N-1$. Finding this dimension was conjectured to be NP-hard. The result presented in this paper considerably strengthens the previously mentioned NP-completeness results and confirms the conjecture that tree embedding is NP-hard. We show that given a tree T and integer k the problem of deciding if T is embeddable in a k-cube such that vertices adjacent in T are also adjacent in the k-cube is NP-complete. A consequence of this result is that it is unlikely there is an efficient algorithm for mapping applications with an irregular communication pattern onto the hypercube. Coincidentally, our reduction uses the same 3-partition problem that was used in [11] to show that the bandwidth minimization problem was NP-complete for binary trees.

2. Terminology and definitions. In the remainder of this paper we adopt the terminology of Garey and Graham [10] and define an n-cube graph to be the undirected Hasse diagram of the lattice on the subsets of a set. Given a set S the **cube on S** is the graph $Q_{|S|} = (V, E)$ where V is the set of all finite subsets of S and an edge (u, v) is in E if and only if the symmetric difference of u and v is a singleton set. The **Hamming distance** between two vertices u, v in the cube on S, $\mathbf{d_H(u, v)}$ is equal to the cardinality of their symmetric difference. The usual n-dimensional $\{0, 1\}$ vector notation of the n-cube can be derived from the previous definition by replacing each subset in the vertex set with its corresponding characteristic vector.

DEFINITION 1. Given $Q_{|S|}$ and a graph $G = (V, E)$, a function mapping the vertices of G to subsets of S is an **embedding** of G in $Q_{|S|}$ if and only if the function is one-one and for all edges $(u, v) \in E(G)$ the Hamming distance between these vertices in $Q_{|S|}$ is one.

The function embedding G in $Q_{|S|}$ is called a **proper labelling** of G. A function is said to be a **partial embedding** of G in an n-cube if the function is a proper labelling on only a partial subgraph of G.

Given a labelling of G in the cube on S, new labellings are obtained by applying rotations and reflections to the old labels of G. A **reflection** of G with respect to u, one of the $2^{|S|}$ subsets of S, is a relabelling of G where the new label of a vertex in G is the symmetric difference of u with the vertex's previous label. A **rotation** of G with respect to one of the $|S|!$ possible permutations of the set S, is a relabelling of G where the new label of a vertex is the label obtained by applying the permutation of S to each of the elements of S in the vertex's old label. The terms rotation and renaming will be used synonymously.

DEFINITION 2. In $Q_{|S|}$ the **neighbourhood**, $\mathbf{N_Q(V')}$ of V', where $V' \subseteq V(Q_{|S|})$, is the set of all vertices in $Q_{|S|}$ (including V') adjacent to a vertex in V'.

Note that the neighbourhood of a vertex u in the cube on S, $N_Q(u)$, is the set of all subsets of S obtained by deleting an element of S from u or adding an element of S not in u to the set u. In a rooted tree, $T = (V, E)$, the neighbourhood of a set of vertices $V' \subseteq V(T)$, $\mathbf{N_C(V')}$ is the set of all children of V'. The degree of vertices in T is taken to be its down-degree (i.e., $\forall v \in V(T), \deg(v) = |N_C(v)|$).

The problem studied in this paper can be formally stated as:

Tree-Embedding: Given a tree T and integer k can T be embedded in Q_k?

We reduce a modified version of the 3-partition problem, which we call 3′-partition, to the tree-embedding problem. The NP-completeness of tree-embedding will follow from the fact that the 3-partition problem, even when expressed in unary, is NP-complete [12]. The definition of the 3-partition problem is

3-Partition: Given a set $A = \{a_i | 0 < i \leq 3m\}$, an integer bound B, and a weight function $s : A \to Z^+$ such that $B/4 < s(a_i) < B/2$ and $\sum_{a \in A} s(a) = mB$. Can A be partitioned into m disjoint 3-element sets $A_i = \{a_{i_0}, a_{i_1}, a_{i_2}\}, 1 \leq i \leq m$ such that $s(a_{i_0}) + s(a_{i_1}) + s(a_{i_2}) = B$?

while the modified version that we use is

3′-Partition: Given a set $A = \{a_i | 0 < i \leq 3m\}$, an integer bound B greater than or equal to $3m$, and a weight function $s : A \to Z^+$ such that

(1) $s(a_{3m-2}) = s(a_{3m-1}) = s(a_{3m}) = 0$,
(2) $B/4 < s(a_i) < B/2$ for all a_i where $i \neq 3m-2, 3m-1, 3m$, and
(3) $\sum_{a \in A} s(a) = (m-1)B$.

Can A be partitioned into $m-1$ disjoint 3-element sets $A_i = \{a_{i_0}, a_{i_1}, a_{i_2}\}$, $1 \leq i < m$ such that $s(a_{i_0}) + s(a_{i_1}) + s(a_{i_2}) = B$?

It is easy to show that these two problems are equivalent. First, note that in 3′-partition the three weightless elements cannot be part of any 3-element subset of A if the weight of the latter is to sum to B. Thus, this is simply an instance of 3-partition on the set $A \backslash \{a_{3m-2}, a_{3m-1}, a_{3m}\}$ with the added restriction that $B > 3m$. The three weightless elements have been introduced for technical reasons that are

explained later. The restriction that $B \geq 3m$ can be satisfied by a simple linear scaling of the weight function. Again this does not change the NP-completeness status of the original problem. This last restriction is introduced because of the way in which both m and B are encoded in the tree we construct.

3. Overview of the reduction. Given an instance of the 3′-partition problem we set $k = B + m + 2$ and construct a tree T that is embeddable in Q_k if and only if there is a solution to the corresponding 3′-partition problem. The tree consists of two parts: a particular part, which encodes an instance of the 3′-partition problem; and a generic part, which encodes the condition that A must be partitioned into $m-1$ sets whose weights sum to B. It is a conflict between these two parts of T that solves the corresponding 3′-partition problem.

This conflict is created by insisting that the generic and particular parts compete for a common set of labels in any proper labelling of T. Vertices in the generic part of T use most of these labels, but any embedding of T in the cube leaves free exactly $m-1$ groups of B labels. In the particular part of T there is, for each $a \in A$, a fan of size $s(a)$ whose labels must come from one of the $m-1$ group of labels left free by the generic part. Again, it is this conflict that must be resolved if there is to exist a solution to the corresponding 3′-partition problem.

The following terminology is introduced to describe the labels that are assigned to the vertices of T. Define Q_{B+m+2} as the cube on S where $|S| = B+m+2$. In addition, let S be the disjoint union of three sets X, Y, and Z where $X = \{x_1, x_2, \cdots, x_m\}$, $Y = \{y_1, y_2, \cdots, y_B\}$, and $Z = \{z_1, z_2\}$. In general the sets X, Y, and Z serve to mark those parts of the cube in which we are interested. Informally, elements of X identify the m sets that partition A, elements of Y express, in unary, the values of the weight function s, and the elements of Z are used to bind the different parts of the tree together.

Given that $S = X \cup Y \cup Z$, labels in the cube on S are subsets of S containing elements of X, Y, and Z. Often however, it is not individual labels that need to be described but rather sets of labels. A set of labels, or equivalently a subset of the power set of S, is called a **form**. In the cube on S, forms are denoted as strings where uppercase letters X, Y, and Z denote arbitrary elements of their respective set and lowercase subscripted letters denote specific elements. For example,

$$XYYYZ \equiv \left\{\begin{array}{r}\text{All subsets of } S \text{ containing 1 element of } X, \text{ 3 elements of } Y,\\ \text{and 1 element of } Z.\end{array}\right\},$$

$$x_3XYYz_1 \equiv \left\{\begin{array}{l}\text{All subsets of } S \text{ containing 2 elements of } X, \text{ where one of the}\\ \text{elements is } x_3, \text{ 2 elements of } Y, \text{ and the element } z_1 \text{ from } Z.\end{array}\right\},$$

$$x_1x_3y_2y_4z_1 \equiv \left\{\begin{array}{r}\text{The subset of } S \text{ containing exactly the elements } x_1,\ x_3,\ y_2,\\ y_4,\ z_1.\end{array}\right\}.$$

Note that forms must correspond to legitimate subsets of S. Thus strings with too many X, Y, or Z letters or strings with elements occurring more than once are not valid forms. The empty set, $\emptyset$ is a legitimate form and denotes the subset of S having no X, Y, or Z elements. The set X' is defined to be the subset $\{x_1, x_2, \cdots, x_{m-1}\}$ of X; X' can also appear in forms.

Furthermore, we define an ordering $<$ on S so that

$$x_1 < x_2 \cdots < x_m < y_1 < y_2 \cdots < y_B < z_1 < z_2.$$

This ordering is extended to a lexigraphical ordering of the subsets of S, where for instance

$$\emptyset < \{x_1, x_2\} < \{x_1, x_2, y_3, z_1\} < \{x_1, x_2, y_4, z_1\}.$$

Subscripted variables can appear in forms and denote a range of elements in X, Y, and Z. For example

$$x_3 x_j[j > 5] \equiv \left\{\begin{array}{l}\text{All subsets of } S \text{ containing 2 elements of } X \text{ where one element} \\ \text{is } x_3 \text{ and the second is an element in the range 6 to } m.\end{array}\right\}.$$

Square brackets appearing at the end of a form or set of forms enclose a restriction on the labels of that form.

Note that with respect to a form F it is still easy to determine the form of the labels in the neighbourhood of F. Recall from § 2 that the neighbourhood of a label α in the cube on S can be obtained by adding or deleting an element from the set α. There is a similar operation that can be performed on strings representing forms. Given a form F one obtains a new form by adding or deleting a single letter from the string denoting F. The set of labels in this new form all have the property that they are Hamming distance one (i.e., adjacent in Q_{B+m+2}) from some label in F. For example, if X is added to the form XYY, then we obtain $XXYY$ where each label in $XXYY$ is adjacent to one or more labels in XYY, that is $XXYY \subseteq N_Q(XYY)$.

There are three labelling functions used to describe different parts of the reduction. Initially, T is constructed with the help of a labelling function l_c that gives a partial embedding of T in the cube. This function is not only used to construct T but also appears in later sections to specify vertices of T indirectly as $l_c^{-1}(\alpha)$ for some label α. In general, whenever l is a labelling and F a form, $l^{-1}(F)$ is used to specify those vertices in T that, by l, are assigned labels from F. The remaining two functions are: l_p, a proper labelling of T that exists when there is a solution to the 3′-partition problem, and l_q, a proper labelling of T that gives rise to a solution to the 3′-partition problem.

Finally we give an outline of the NP-completeness proof for the tree-embedding problem.

THEOREM 3.1. *Tree-embedding is* NP*-complete.*

Outline of Proof. The tree-embedding problem is in NP since, given a labelling, one can check in polynomial time that the Hamming distance between adjacent vertices in T is one and that the labels are distinct. Now, given an instance of the 3′-partition problem we set $k = B + m + 2$ and construct the tree T. The proof is complete if the following two statements are true.

(1) Given a solution of 3′-partition there exists a proper labelling, l_p, embedding T in Q_{B+m+2}, and
(2) Conversely, given an embedding of T in Q_{B+m+2}, that is a proper labelling l_q, then it is possible to extract from the labelling of T a solution to the 3′-partition problem.

The construction of T is given in the next section. The proof of the first statement is given by Theorem 5.3 in § 5 and the second statement is established by Theorem 6.4 in § 6.

4. Construction. We begin by constructing from an instance of 3′-partition a tree T encoding both the constituents of the instance and also the constraints that must be satisfied. As previously mentioned, most of T will be defined with the help of

the labelling function l_c. This approach is taken since, up to a renaming of elements within the sets X, Y, and Z, many of the vertices of T are assigned the same label in any proper labelling of T. The overall shape of T is such that, except for those vertices that create the conflict between the generic and particular parts of T, there is only one fixed embedding.

A skeleton of the tree T is given in Fig. 1. Each box in this skeleton represents

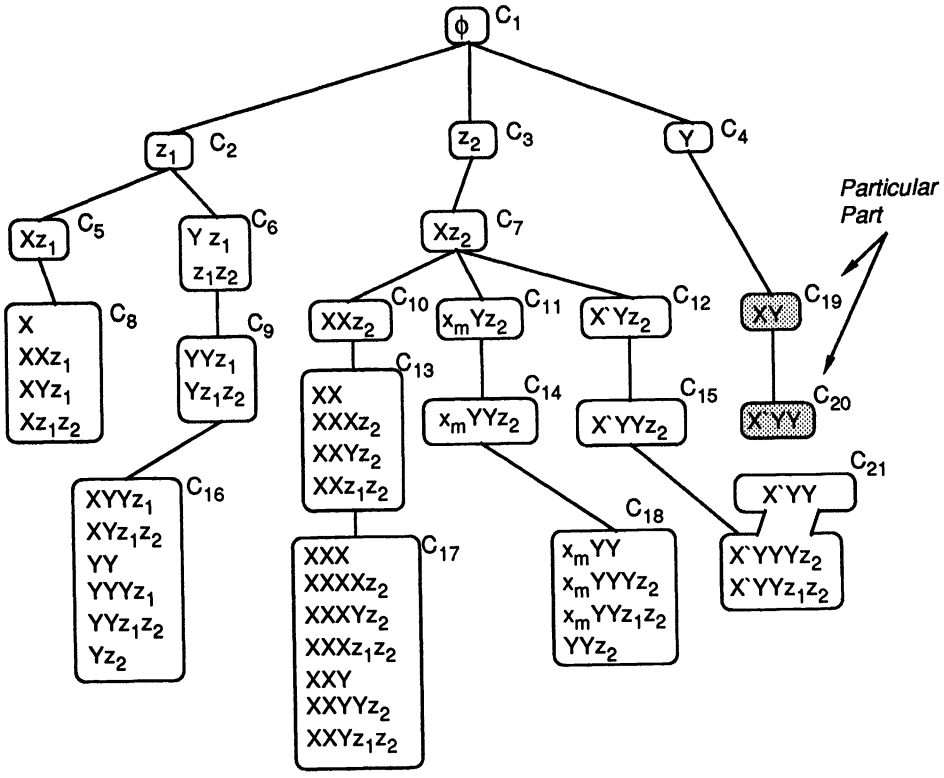


FIG. 1. *Skeleton of T where $X = \{x_1, x_2, \cdots, x_m\}, Y = \{y_1, y_2, \cdots, y_B\}, Z = \{z_1, z_2\}$, and $X' = \{x_1, x_2, \cdots, x_{m-1}\}$ of X.*

one or more vertices of T on a particular level of the tree. Edges denote some, as yet unspecified, connection between the vertices of T on one level and their children on the next level.

The construction of T from its skeleton is completed in two stages. First, the edges and vertices of T in C_1 to C_{18} are defined. The vertices are defined by the labelling function l_c; there is a vertex in T for each label belonging to a form inside C_1 to C_{18}. Second, the remaining portion of the tree (C_{19}, C_{20}, and C_{21}) is defined by explicitly describing its structure. It is in this part of the tree that we encode the 3′-partition problem. The forms that appear inside C_{19}, C_{20}, and C_{21} of Fig. 1 are the labels of these vertices, should a solution to the problem exist.

Note that vertices in C_{20} and vertices in C_{21} both receive labels of the form $X'YY$. It is this common set of labels that is the source of the conflict between the generic and particular parts of T. The generic part of T has been carefully constructed so that,

in any embedding of T in Q_{B+m+2}, it surrounds the particular part of T leaving free, for each $i \neq m$, B labels of the form x_iYY. The leaves of each fan in the particular part of T can only be assigned labels from *one* of the $m-1$ groups of free labels. Again, T can be properly labelled, if and only if, the $3(m-1)$ fans in C_{19} and C_{20} can cover all of the $m-1$ labels left free by the generic part of T.

The following three sections describe the construction of T from its skeleton. Section 4.1 defines those edges connecting vertices in C_1 to C_{18}. Section 4.2 describes the vertices and edges in the particular part of the tree, C_{19} and C_{20}. Finally, § 4.3 describes the vertices in C_{21} and the edges that connect them to their parents in C_{15}.

4.1. Construction of C_1 to C_{18}. Let the vertices in C_1 to C_{18} be defined by the labels given in Fig. 1 (i.e., $l_c(C_1) = \emptyset$; $l_c(C_2) = z_1; \cdots; l_c(C_6) = Yz_1, z_1z_2$; etc.). Given a lexigraphical ordering of the subsets of S define the edge set between the vertices in C_i and children in C_j by introducing a function f that maps a vertex u in C_j to the vertex in C_i whose label is the **least** label Hamming distance one from the label of u.

Formally, given C_i and C_j in the skeleton of T such that $i, j \leq 18$ and C_i is the parent of C_j then let f be the function

$$f : C_j \longrightarrow C_i$$

where $f(u) = v$ such that

(1) $d_H(l_c(u), l_c(v)) = 1$, and

(2) $\forall v' \in C_i$ if $d_H(l_c(v'), l_c(u)) = 1$ then $l_c(v) \leq l_c(v')$.

There is an edge $(u, v) \in E(T)$ if and only if $f(u) = v$. For example, the connections between $C_6 = l_c^{-1}(Yz_1, z_1z_2)$ and its children in $C_9 = l_c^{-1}(YYz_1, Yz_1z_2)$, for $B = 4$, is given in Fig. 2. There is an edge shown in Fig. 2 whenever the Hamming distance between two labels is one, the solid edges belong to T. Observe that the degree of

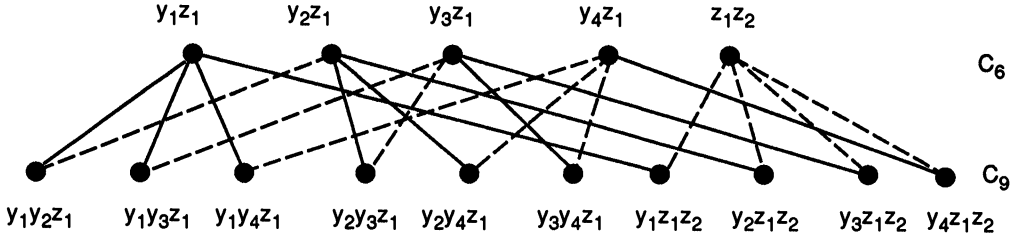


FIG. 2. *An example of the tree connections for $B = 4$, where the solid lines are edges in T.*

the vertices in C_6 decrease. This type of structure appears throughout T and helps fix the embedding of T in Q_{B+m+2}.

LEMMA 4.1. *The part of T whose vertices are defined by l_c and whose edges are defined by f is a tree.*

Proof. Consider any two nodes C_i and C_j labelled by l_c such that C_i is the parent of C_j in the skeleton of T. Note that every form F in $l_c(C_j)$ can be obtained from some form F' in $l_c(C_i)$ by adding or deleting an X, Y, or Z element from the string denoting F'. Thus $F \subseteq N_Q(F')$, and for every $l_c(u) \in F$ there exists a $l_c(v) \in F'$ Hamming distance one away. This and the fact that S is well-ordered implies that f is a well-defined function on all of the labels in $l_c(C_j)$.

Since this holds for all C_i, C_j pairs labelled by l_c, the graph T that was constructed is connected and on each level there is only one edge joining a vertex in C_j to a vertex in C_i on the level above. Thus the part of T constructed by l_c and f is a tree. □

4.2. Construction of C_{19} to C_{20}. Root from each of the vertices $l_c^{-1}(y_1)\cdots l_c^{-1}(y_B)$ in C_4 a fan of size m. Now for each $a_i \in A$ such that $s(a_i) > 0$ uniquely choose one of the fans rooted at $l_c^{-1}(y_1)$, $l_c^{-1}(y_2), \cdots, l_c^{-1}(y_{3(m-1)})$ and root at a leaf in that fan a second fan of size $s(a_i)$ (see Fig. 3).

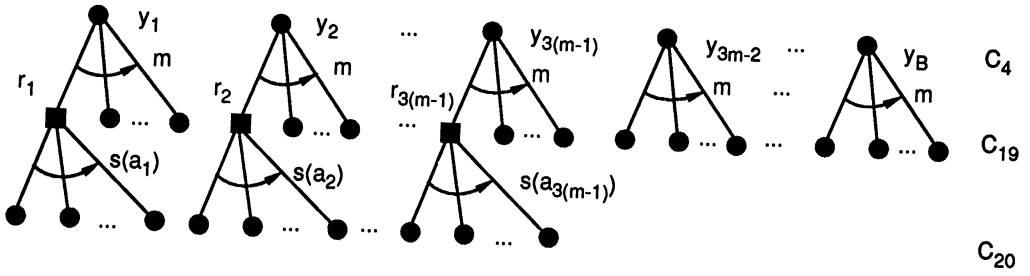


FIG. 3. *Encoding of the instance of 3′-partition*

DEFINITION 3. Let $\mathbf{r_i}$ be the vertex in C_{19}, corresponding to the element $a_i \in A$, where a fan of size $s(a_i) > 0$ was rooted. Let $\mathbf{R}$ denote the set of all such vertices in C_{19}.

Note that for the three weightless elements of A $(a_{3m-2}, a_{3m-1}, a_{3m})$ we have implicitly rooted, at y_{3m-2}, y_{3m-1}, and y_{3m}, a fan of size zero (i.e., no children). This is consistent with the fact that all labels of the form $x_m YY$ are assigned to vertices in C_{18}, and thus there are no free labels available for one of the m sets in the partition. Also note that there is a direct correspondence between the set of vertices R and the set $A\backslash\{a_{3m-2}, a_{3m-1}, a_{3m}\}$. When there is an embedding of T in Q_{B+m+2} it will be the X element in r_i's label that identifies the 3-element set in the partition to which r_i (and thus a_i) belongs.

4.3. Construction of C_{21}. The remaining vertices of T are all children of vertices in C_{15}. Given is a rather cryptic description of the children of C_{15}, but one that simplifies the definition of l_p in the next part of the proof. At this stage it suffices to verify that T can be constructed.

Given that $l_c(C_{15}) = X'YYz_2$, consider an arbitrary vertex of the form $l_c^{-1}($ $x_i y_j y_k z_2$ $[i \neq m]$) and, without loss of generality, assume that $j < k$. Define, with respect to index i, the interval $I = \{3i-2, 3i-1, 3i\}$. The degrees of the vertices in C_{15} will depend on the values of their indices i, j, and k, and whether or not they are contained in I.

For an arbitrary vertex $u = l_c^{-1}(x_i y_j y_k z_2[i \neq m])$:

(1) If $j, k \notin I$ then
  (a) if $k < 3i$ then $\deg(u) = B - k + 2$,
  (b) otherwise $\deg(u) = B - k + 5$.
(2) If j or $k \in I$ but not both then
  (a) if $j = 3i - 2$ or $k = 3i - 2$ then $\deg(u) = 2$,
  (b) otherwise $\deg(u) = 3$.
(3) If $j, k \in I$ then
  (a) if $j \neq 3i - 2$ and $k \neq 3i - 2$ then $\deg(u) = 2$,
  (b) otherwise $\deg(u) = 1$.

Now add to T the number of children needed to satisfy the degree requirements just given for the vertices in C_{15}. The degree of all vertices in C_{15} is well-defined since the label of each vertex in C_{15} has either 0, 1, or 2 indices from I.

This completes the construction of T. Since the forms that appear in the skeleton of T have constant length, the size of T is polynomial in B and m. In the next section we show that if there is a solution to the instance of 3′-partition then T is embeddable in Q_{B+m+2}.

4.4. Properties of T. Before proving, in §§ 5 and 6, that tree embedding is NP-complete we need the following lemmas. First we prove, as claimed, that the labelling function l_c is a partial embedding of T in Q_{B+m+2}.

LEMMA 4.2. *The function l_c is a partial embedding of T in Q_{B+m+2}.*

Proof. Function l_c is a partial embedding if for the part of T on which l_c is defined, the Hamming distance between adjacent vertices is one and the labels are distinct. The Hamming distance is one since, by function f in § 4.1, this is precisely how the edges in C_1 to C_{18} were defined. Also, by checking the forms in Fig. 1 it is easily verified that all of the labels are distinct. □

The following two lemmas are used in § 6.2 and establish some facts about the structure of T at C_5 and C_{10}.

LEMMA 4.3. *For all* $1 \leq i \leq m$, $\deg(l_c^{-1}(x_i z_1)) = m + B - i + 2$.

Proof. In the construction of T the labels of the form Xz_1 were connected to labels of the form $\{X, XXz_1, XYz_1, Xz_1z_2\}$. Recall from the construction of T the function f that defined the edges between vertices in C_5 and C_8. There was an edge joining $l_c^{-1}(x_i z_1)$ to a vertex $u \in C_8$ whenever $x_i z_1$ was the least label in C_5 Hamming distance one from $l_c(u)$. Therefore, vertex $l_c^{-1}(x_i z_1)$ has

$m - i$	children of the form	$x_i x_j z_1 [i < j]$,
B	children of the form	$x_i Y z_1$,
1	child of the form	x_i,
1	child of the form	$x_i z_1 z_2$.

Thus, $\deg(l_c^{-1}(x_i z_1)) = m + B - i + 2$. □

We now determine the degree of the vertices at C_{10}.

LEMMA 4.4. *For all* $1 \leq i < j \leq m$, $\deg(l_c^{-1}(x_i x_j z_2)) = m + B - j + 2$.

Proof. In the construction of T, the labels in $l_c(C_{10}) = XXz_2$ are joined to labels in $l_c(C_{13}) = \{XX, XXXz_2, XXYz_2, XXz_1z_2\}$. By function f, there is an edge joining $l_c^{-1}(x_i x_j z_2)$ to a vertex in C_{13} whenever $x_i x_j z_2$ is the least label a Hamming distance one from a label in $l_c(C_{13})$. Therefore, there is an edge connecting $l_c^{-1}(x_i x_j z_2)$ to

$m - j$	children of the form	$x_i x_j x_k z_2 [j < k]$,
B	children of the form	$x_i x_j Y z_2$,
1	child of the form	$x_i x_j$,
1	child of the form	$x_i x_j z_1 z_2$.

Thus $\deg(l_c^{-1}(x_i x_j z_2)) = m + B - j + 2$. □

5. Embedding a solution of 3′-partition in Q_{B+m+2}. Suppose A can be partitioned into $m - 1$ sets $\{a_{i_0}, a_{i_1}, a_{i_2}\}$, $1 \leq i < m$, satisfying the conditions of the 3′-partition problem. We shall show that T is embeddable in Q_{B+m+2}. That is, there exists a function l_p labelling T with subsets of the set S such that

(1) The Hamming distance between the labels of vertices adjacent in T is one (i.e., for all $(u,v) \in E(T)$, $d_H(l_p(u), l_p(v)) = 1$), and
(2) The labels are distinct.

The function l_p is defined incrementally in three steps. First, from l_c, a new labelling function l_p'' is defined and is shown to be a partial embedding of $C_1, \cdots, C_{18}$ in Q_{B+m+2}. Then, we extend l_p'' to l_p', a partial embedding of T that includes the vertices in C_{19}. Finally, we define l_p, a proper labelling of all of T, by extending l_p' to include the vertices in C_{20} and C_{21}.

5.1. Embedding C_1 to C_{18} in Q_{B+m+2}. First, let a_{i_j} or equivalently r_{i_j}, $j = 0,1,2$ and $1 \leq i < m$, denote the jth element in the ith set of the partition of A. Recall from the construction of T that there is, for each a_k in A ($s(a_k) > 0$), a fan of size $s(a_k)$ rooted at r_k in R and that the parent of r_k in C_4 is $l_c^{-1}(y_k)$. We relabel the vertices of C_4 so that the parent of r_{i_j} is now assigned the label y_{3i+j-2}. This renames the vertices of C_4 so that the i^{th} 3-element set in the partition of A now corresponds to the three consecutive labels y_{3i-2}, y_{3i-1}, y_{3i}.

Formally, define a permutation π, such that if the parent of r_{i_j} is $l_c^{-1}(y_k)$, then

$$\pi(y_k) = y_{3i+j-2}.$$

Now, with respect to π, define l_p'' to be the following function on the vertices in $C_1, \cdots, C_{18}$

$$l_p''(v) = \begin{cases} \pi(l_c(v)) & \text{if } v \in C_4, \\ l_c(v) & \text{otherwise.} \end{cases}$$

LEMMA 5.1. *The function l_p'' is a partial embedding of T in Q_{B+m+2}.*

Proof. The Hamming distance between adjacent vertices in $\{C_1, \cdots, C_{18}\} \backslash C_4$ is one; since, by Lemma 4.2, l_c is a partial embedding of T on those vertices. Also, since $l_p''(C_1) = \emptyset$ and $l_p''(C_4) = Y$, the Hamming distance between $\emptyset$ and the labels of vertices in C_4 is one. Finally, since π is a permutation of only the Y elements in $l_c(C_4)$ the labels assigned by l_p'' remain distinct. Thus, l_p'' is a partial embedding of T in Q_{B+m+2}. □

5.2. Adding C_{19} to the embedding. Now we extend l_p'' to label the vertices in C_{19}. Recall, from the construction of T, that in C_{19} there are B groups of m vertices, and that all the vertices in each group have the same parent in C_4. Moreover, by the reordering of the Y's in the last step, each $r_{i_j} \in R \subseteq C_{19}$ is the child of the vertex in C_4 whose label under l_p'' is y_{3i+j-2}. The vertices in C_{19} will be assigned labels of the form XY.

First, we assign to each group of m vertices whose parent is labelled y_k labels of the form Xy_k. Within a group these m labels can be assigned arbitrarily except that r_{i_j}, if there is one, must be labelled $x_i y_k$.

In summary, we have for $v \notin C_{19}$

$$l_p'(v) = l_p''(v)$$

and for $v \in C_{19}$

$$l_p'(N_C(l_p''^{-1}(y_k))) = Xy_k$$

where $l'_p(r_{i_j}) = x_i y_k$ for $k = 3i + j - 2$. Note that, as promised, the index of the X element of the r_{i_j}'s in R identifies the 3-element set to which r_{i_j} belongs.

LEMMA 5.2. *The function l'_p is a partial embedding of T into Q_{B+m+2}.*

Proof. Again we can easily check that l'_p is a proper labelling of C_1 to C_{19}. It suffices to check only the labels of vertices in C_{19} since by l''_p the remaining vertices are already properly labelled. The labels of the form XY are distinct from those in the rest of T and the mB labels of this form equal the number of vertices in C_{19}. Second, since the m children of vertex $l''_p(y_k)$ in C_4 are assigned labels of the form Xy_k, the Hamming distance between adjacent vertices is one. Therefore l'_p is a proper labelling of $C_1, \cdots, C_{19}$ in Q_{B+m+2}. □

5.3. Adding C_{20} and C_{21} to the embedding. As mentioned previously, labels of the form $X'YY$ are assigned to vertices in both C_{20} and C_{21}, and so we must ensure that the labels assigned to these two sets of vertices do not conflict.

Define the following two sets

$$\begin{aligned} P_i &= \{N_C(r_{i_0}) \cup N_C(r_{i_1}) \cup N_C(r_{i_2})\}, \text{ and} \\ G_i &= N_C(l_c^{-1}(x_i YYz_2)). \end{aligned}$$

The set P_i corresponds to a 3-element set in the particular part of T and G_i corresponds to a set of vertices in the generic part of T. Furthermore,

$$C_{20} = \bigcup_{1 \le i < m} P_i \text{ and } C_{21} = \bigcup_{1 \le i < m} G_i.$$

Now, for each of the $m - 1$ pairs of sets P_i and G_i, vertices in P_i will be assigned labels of the form $x_i YY$ and vertices in G_i will be assigned labels of the form $\{x_i YY$, $x_i YYYz_2$, $x_i YYz_1z_2\}$. These labels are assigned so that those $x_i YY$'s used in G_i are distinct from those used in P_i.

In general, define

$$l_p(v) = l'_p(v), \qquad v \notin C_{20}, C_{21}$$

and consider, for all i, the two sets of vertices P_i and G_i. In the previous section l'_p was defined so that $l'_p(r_{i_j}) = x_i y_{3i+j-2}$. Given that $I = \{3i - 2, 3i - 1, 3i\}$ let the set

$$Y \backslash \{y_{3i-2}, y_{3i-1}, y_{3i}\} = \{y_{k_1}, y_{k_2}, \cdots, y_{k_{B-3}}\}$$

and let

$$l_p(N_C(r_{i_0})) = \begin{cases} x_i y_{3i-2} y_{k_l} & \text{where } 1 \le l \le s(a_{i_0}) - 1, \\ x_i y_{3i-2} y_{3i-1}, \end{cases}$$

$$l_p(N_C(r_{i_1})) = \begin{cases} x_i y_{3i-1} y_{k_l}, & \text{where } s(a_{i_0}) - 1 < l \le s(a_{i_0}) + s(a_{i_1}) - 2, \\ x_i y_{3i-1} y_{3i}, \end{cases}$$

$$l_p(N_C(r_{i_2})) = \begin{cases} x_i y_{3i} y_{k_l}, & \text{where } s(a_{i_0}) + s(a_{i_1}) - 2 < l \le B - 3, \\ x_i y_{3i} y_{3i-2}. \end{cases}$$

First, note that

$$\forall u \in N_C(r_{i_j}), \quad d_H(l_p(u), x_i y_{3i+j-2}) = 1.$$

Second, with respect to I and $N_C(r_{i_j})$, note that $s(a_{i_j}) - 1$ of the vertices are assigned labels with one Y element whose index is in I and that the last vertex is assigned a

label with both Y elements having indices in I. Therefore, in total, $B-3$ of the labels have one index from I and the remaining three labels contain Y elements with both indices in I. This interval is exactly the same one used in § 4.3 to construct G_i and it allows us to easily distinguish between the x_iYY's used by P_i and the remaining ones that can be used by G_i.

Now, since $l_c(v) = l_p''(v) = l_p'(v) = l_p(v)$ for $v \in C_{15}$, the vertices in G_i are the children of the vertices $l_c^{-1}(x_iYYz_2)$ in C_{15}. The degree of these vertices was given in § 4.3 so it is only necessary to match labels with the corresponding degrees of labels in C_{15}.

Consider an arbitrary element $l_c^{-1}(x_iy_jy_kz_2)$ where $i \neq m$.

(1) If $j, k \notin I$ (without loss of generality assume that $j < k$), then if $k < 3i$ label the

$$\text{children of } x_iy_jy_kz_2 \text{ with } \begin{cases} x_iy_jy_kz_1z_2, \\ x_iy_jy_k, \\ x_iy_jy_ky_lz_2 & \text{where } l > k, \end{cases}$$

otherwise label the

$$\text{children of } x_iy_jy_kz_2 \text{ with } \begin{cases} x_iy_jy_kz_1z_2, \\ x_iy_jy_k, \\ x_iy_jy_ky_lz_2 & \text{where } l > k, \\ x_iy_jy_ky_{3i-2}z_2, \quad x_iy_jy_ky_{3i-1}z_2, \quad x_iy_jy_ky_{3i}z_2. \end{cases}$$

By the construction of T, if $k < 3i$ then each $l_c^{-1}(x_iy_jy_kz_2)$ has $B-k+2$ children and there are the same number of available labels; one label of the form $x_iy_jy_kz_1z_2$, one label of the form $x_iy_jy_k$, and $B-k$ labels of the form $x_iy_jy_ky_lz_2$. Also, if $k \not< 3i$ then $l_c^{-1}(x_iy_jy_kz_2)$ has $B-k+5$ children. The labels are distinct from those in P_i because both $j, k \notin I$, and as required, the labels of the vertices in C_{15} are all Hamming distance one from the labels of their children. As an aside, note that *all* labels of the form $x_iy_jy_ky_lz_2$ with at most one index in I label vertices in this part of T.

(2) If j or $k \in I$ but not both (without loss of generality assume that $k \in I$), then there are three cases to consider depending on the value of j, an element of

$$Y\backslash\{y_{3i-2}, y_{3i-1}, y_{3i}\} = \{y_{k_1}, y_{k_2}, \cdots, y_{k_{B-3}}\}.$$

(a) If $1 \leq l \leq s(a_{i_0}) - 1$ where $y_j = y_{k_l} \in Y\backslash\{y_{3i-2}, y_{3i-1}, y_{3i}\}$, then

Vertex	No. of children	Labels of children
$x_iy_jy_{3i}z_2$	3	$\mathbf{x_iy_jy_{3i}}$, $x_iy_jy_{3i}y_{3i-1}z_2$, $x_iy_jy_{3i}z_1z_2$
$x_iy_jy_{3i-1}z_2$	3	$\mathbf{x_iy_jy_{3i-1}}$, $x_iy_jy_{3i-1}y_{3i-2}z_2$, $x_iy_jy_{3i-1}z_1z_2$
$x_iy_jy_{3i-2}z_2$	2	$x_iy_jy_{3i}y_{3i-2}z_2$, $x_iy_jy_{3i-2}z_1z_2$

(b) If $s(a_{i_0})-1 < l \leq s(a_{i_0})+s(a_{i_1})-2$ where $y_j = y_{k_l} \in Y\backslash\{y_{3i-2}, y_{3i-1}, y_{3i}\}$, then

Vertex	No. of children	Labels of children
$x_iy_jy_{3i}z_2$	3	$\mathbf{x_iy_jy_{3i}}$, $x_iy_jy_{3i}y_{3i-2}z_2$, $x_iy_jy_{3i}z_1z_2$
$x_iy_jy_{3i-1}z_2$	3	$x_iy_jy_{3i-1}y_{3i-2}z_2$, $x_iy_jy_{3i}y_{3i-1}z_2$, $x_iy_jy_{3i-1}z_1z_2$
$x_iy_jy_{3i-2}z_2$	2	$\mathbf{x_iy_jy_{3i-2}}$, $x_iy_jy_{3i-2}z_1z_2$

(c) If $s(a_{i_0}) + s(a_{i_1}) - 2 < l \leq B - 3$, where $y_j = y_{k_l} \in Y \backslash \{y_{3i-2}, y_{3i-1}, y_{3i}\}$, then

Vertex	No. of children	Labels of children
$x_i y_j y_{3i} z_2$	3	$x_i y_j y_{3i} y_{3i-2} z_2$, $x_i y_j y_{3i} y_{3i-1} z_2$, $x_i y_j y_{3i} z_1 z_2$
$x_i y_j y_{3i-1} z_2$	3	$\mathbf{x_i y_j y_{3i-1}}$, $x_i y_j y_{3i-1} y_{3i-2} z_2$, $x_i y_j y_{3i-1} z_1 z_2$
$x_i y_j y_{3i-2} z_2$	2	$\mathbf{x_i y_j y_{3i-2}}$, $x_i y_j y_{3i-2} z_1 z_2$

Note that they do not conflict with labels assigned in C_{20} since those $x_i YY$ labels used in C_{20} are not used in C_{21} and it is possible to juggle the $x_i YYY z_2$ labels so that each child of $x_i YY z_2$ is labelled. Also in each case the children are Hamming distance one from the label of their parent. Note that all labels of the form $x_i y_j y_k y_l z_2$ with exactly two indices in I label vertices in this part of T.

(3) Finally, if both $j, k \in I$, then let

Vertex	No. of children	Labels of children
$x_i y_{3i} y_{3i-1} z_2$	2	$x_i y_{3i} y_{3i-1} y_{3i-2} z_2$, $x_i y_{3i} y_{3i-1} z_1 z_2$
$x_i y_{3i} y_{3i-2} z_2$	1	$x_i y_{3i} y_{3i-2} z_1 z_2$
$x_i y_{3i-1} y_{3i-2} z_2$	1	$x_i y_{3i-1} y_{3i-2} z_1 z_2$

In each of the above cases the $x_i YY$ element is missing and was used to label the vertices in P_i. Also, the children are all Hamming distance one from the label of their parent. Finally, note that the last remaining label of the form $x_i YYY z_2$ ($x_i y_j y_k y_l z_2$ with indices $j, k, l \in I$) appears as a label in this part of T.

This completes the definition of the final labelling function l_p. Note, as implied by our remarks, every label of the form $\{x_i YY, x_i YYY z_2, x_i YY z_1 z_2\}$ was assigned to a vertex in C_{20} or C_{21}, so that $l_p(C_{20} \cup C_{21}) = \{X'YY, X'YYY z_2, X'YY z_1 z_2\}$. This fact gives us a count of the number of vertices in C_{20} and is used in the second half of the proof to show that there are no "free" labels. That is, any proper labelling of T in Q_{B+m+2} must use exactly, up to reflections and rotations, the same set of labels.

THEOREM 5.3. *The function l_p is an embedding of T into Q_{B+m+2}.*

Proof. The proof of this theorem follows from the remarks made during the definition of l_p that showed that vertices in C_{20} and C_{21} were properly labelled. The fact that l_p is a proper labelling for the rest of T follows directly from Lemma 5.2. □

This proves the first statement in the outline of Theorem 3.1. That is, given a solution to the instance we can construct a proper labelling l_p of T in Q_{B+m+2}.

6. Extracting a 3′-partition from an embedding. Conversely, we must show that if T is embeddable in Q_{B+m+2} then there exists a solution to the corresponding 3′-partition problem. Indeed, we show how to extract, from *any* embedding of T in Q_{B+m+2}, a solution to the instance. A solution can be extracted because, up to a rotation of the cube, the embedding of T in Q_{B+m+2} is unique and as a result the conflict described earlier in §§ 4 and 5.3 must be resolved by the embedding. As before, a resolution of this conflict must construct, indirectly, a partition of A satisfying the conditions of 3′-partition.

Suppose that T has been embedded in Q_{B+m+2}. First, this embedding is put

into a regular form by performing the following transformations, we take l_q to be the resulting label function.

(1) Apply a reflection to T so that l_q of the root is $\emptyset$.

(2) Apply a rotation to T so that in the top levels of the tree $l_q(C_2) = z_1$, $l_q(C_3) = z_2$, and $l_q(C_4) = Y$.

Once the labels of C_7 have been determined we will also permute the elements of X so that the parent in C_7 of the vertices in C_{11} are labelled $x_m z_2$. This is not necessary, however by doing so, the form of the tree under l_q becomes identical to the form of T depicted in Fig. 1.

The remainder of this section is divided into two parts. First, it is shown that the form of the labels assigned to the skeleton of T under l_q are the same as those in Fig. 1. This now includes the labels in C_{19}, C_{20}, and C_{21} so that $l_q(C_{19}) = XY$ and that taken together $l_q(C_{20} \cup C_{21}) = \{X'YY, X'YYYz_2, X'YYz_1z_2\}$. In the second part, the fact that $X'YY \subseteq l_q(C_{20} \cup C_{21})$, is used to extract a solution to the instance of 3'-partition.

6.1. Determining the form of T. We show that *any* embedding of T in Q_{B+m+2} assigns labels of the form depicted in Fig. 1 to the vertices in T. More specifically, for any embedding l_q,

(1) $l_q(C_1) = l_c(C_1)$, $l_q(C_2) = l_c(C_2)$, $\cdots$, $l_q(C_{18}) = l_c(C_{18})$,

(2) $l_q(C_{19}) = XY$, and

(3) $l_q(C_{20} \cup C_{21}) = \{X'YY,\ X'YYYz_2,\ X'YYz_1z_2\}$.

Statements such as $l_q(C_1) = l_c(C_1)$ specify that, up to a permutation of labels within a node of the skeleton, the functions l_q and l_c are equivalent.

In order to extract a solution to 3'-partition it would suffice to know $l_q(C_{20} \cup C_{21})$. However the forms assigned to these vertices cannot be determined until the forms in the remaining parts of the tree are known. It is not necessary to determine precisely the label of each vertex in T since the sole purpose of most of T is simply to consume labels in Q_{B+m+2}, thereby reducing the labels that could have been assigned to vertices in C_{20} and C_{21}.

The first step in determining the form of T is to describe the conditions under which the form of a node in the skeleton is forced. The following argument, which we call *label forcing*, is used repeatedly to determine the form of vertices in T.

Let $\mathcal{P}(S)$ denote the power set of S, the set of all labels in Q_{B+m+2}. Also, if F and G are forms, then let $\mathbf{N_F(G)}$ denote $N_Q(G) \cap F$, the neighbourhood of G restricted to F. Now suppose, for a set of vertices $V' \subseteq V(T)$, that $l_q(V')$ has already been determined. We would like, for some set of vertices $W \subseteq V'$, to determine the labels of $N_C(W)$ where it is assumed that the vertices in $N_C(W)$ are disjoint from V'. The labels assigned to $N_C(W)$ must satisfy the following two conditions. First, since l_q is a proper labelling of T, the Hamming distance between the label of a vertex in W and its child in $N_C(W)$ must be one. Hence, $l_q(N_C(W)) \subseteq N_Q(l_q(W))$. Second, since $l_q(V')$ has already been determined, $l_q(N_C(W)) \subseteq \mathcal{P}(S) - l_q(V')$. Clearly, if $F = \mathcal{P}(S) - l_q(V')$, then from our previous remarks, $l_q(N_C(W)) \subseteq N_F(l_q(W))$. (In the remaining sections, F or F' denotes the set of free labels, those labels whose vertices have yet to be determined.) Finally, if in addition to the last two conditions, the number of available labels exactly equals the number of vertices to be labelled, then $l_q(N_C(W)) = N_F(l_q(W))$. In summary,

Label Forcing: If

(1) $l_q(W)$, are the labels of a set of vertices W in T,

(2) F is the set of free labels, and

(3) $|N_C(W)| = |N_F(l_q(W))|$
then $l_q(N_C(W)) = N_F(l_q(W))$.

The general strategy for determining the form of T is to start with the root and its children, whose forms are already known, and to deduce, by label forcing, the form of their children by examining the set of free labels. Once the form of these vertices has been determined then, since these forms are no longer available, the form of their children can be deduced. This process is repeated until the form of the entire tree has been determined.

For the most part it is sufficient to look macroscopically at the neighbourhood of a node in the skeleton of T and compare the cardinality of the node with the cardinality of its neighbourhood. However at C_2 and C_7, where we must distinguish between different children, it is necessary to examine, microscopically, the structure of T inside the node. Again the tree T was constructed so that label forcing occurs both macroscopically and microscopically. In the next section we show that with some global information about the form of C_2, C_7 and their children it is possible to distinguish between the labels assigned to different sets of children. In turn, these results are used in the final section to determine macroscopically the form of the skeleton of T.

6.2. Children of C_2 to C_7. The two results given in this section use Lemmas 4.3 and 4.4 from § 4.4 to determine, first, C_5's labels and, second, C_{10}'s labels. Basically, the approach is simply a microscopic version of the strategy outlined in the last section. For each possible label that could be assigned to a vertex in C_5 (C_{10}) we compare the size of its neighbourhood in F, the available labels, with the degree of the vertex. By a process of elimination, since this neighbourhood must always be larger than or equal to the degree, the form of the vertices in C_5 (C_{10}) can be determined.

Let v_i denote $l_c^{-1}(x_iz_1)$ in C_5. Note that the v_i's are those vertices whose degrees are given by Lemma 4.3.

LEMMA 6.1. *If* $l_q(C_5 \cup C_6) = \{Xz_1, Yz_1, z_1z_2\}$ *and* $l_q(C_8 \cup C_9) = \{X, XXz_1, XYz_1, Xz_1z_2, YYz_1, Yz_1z_2\}$, *then* $l_q(C_5) = Xz_1$, $l_q(C_6) = \{Yz_1, z_1z_2\}$, $l_q(C_8) = \{X, XXz_1, XYz_1, Xz_1z_2\}$, *and* $l_q(C_9) = \{YYz_1, Yz_1z_2\}$.

Proof. The proof is by induction on the vertices in C_5. Let $F = l_q(C_8 \cup C_9) = \{X, XXz_1, XYz_1, Xz_1z_2, YYz_1, Yz_1z_2\}$; we claim that for all i the following property holds:

$$\mathrm{P}(i): \quad l_q(v_i) \in Xz_1 \text{ and } l_q(N_C(v_1, v_2, \cdots, v_i)) = N_F(l_q(v_1, v_2, \cdots, v_i)).$$

The fact that P(1) holds follows from the same argument given for the general case and is left to the reader. So suppose that P(j) holds for all $j < i$, and assume, without loss of generality, that $l_q(v_j) = x_jz_1$.

First, let us determine the set of free labels that can be used to label the children of v_i. The set of free labels consists of those labels that remain after removing from F the labels assigned to vertices in $N_C(v_1, v_2, \cdots, v_{i-1})$. Let F' denote these labels where, by the inductive hypothesis,

$$\begin{aligned} F' &= F - l_q(N_C(v_1, v_2, \cdots, v_{i-1})) = F - N_F(l_q(v_1, v_2, \cdots, v_{i-1})) \\ &= F - N_F(x_jz_1[j < i]) = F - \{x_j,\ x_jXz_1,\ x_jYz_1,\ x_jz_1z_2\}[j < i] \\ &= \{x_j,\ x_jx_kz_1,\ x_jYz_1,\ x_jz_1z_2,\ YYz_1,\ Yz_1z_2\}[j, k \geq i]. \end{aligned}$$

By Lemma 4.3, v_i has $m + B - i + 2$ children whose labels must come from F', where by hypothesis the label of v_i is of the form Xz_1, Yz_1, or z_1z_2. However, label

z_1z_2 has only $m+B-i+1$ neighbours in F', $m-i+1$ of the form $x_jz_1z_2[j \geq i]$ and B of the form Yz_1z_2. Labels of the form Yz_1 also have only $m+B-i+1$ neighbours in F'; $m-i+1$ labels of the form $x_jYz_1[j \geq i]$, $B-1$ labels of the form YYz_1, and 1 label of the form Yz_1z_2. Thus, $l_q(v_i)$ is in Xz_1 and equals for some $j \geq i$, x_jz_1, which has exactly $m+B-i+2$ neighbours in F'; $m-i$ labels of the form $x_jx_kz_1[j,k \geq i]$, B labels of the form x_jYz_1, 1 label of the form x_j, and 1 label of the form $x_jz_1z_2$.

Now, since the degree of v_i exactly equals the number of available labels, it follows that all of these labels are used to label the children of v_i, that is, $l_q(N_C(v_i)) = N_{F'}(l_q(v_i))$. Finally, if these labels are added to those labels already known to be assigned to vertices in $N_C(v_1, v_2, \cdots, v_{i-1})$, then we have that $l_q(N_C(v_1, v_2, \cdots, v_i)) = N_F(l_q(v_1, v_2, \cdots, v_i))$. Hence P($i$) holds.

Therefore, by the principle of induction, P(i) is true for all i which implies that $l_q(C_5) = Xz_1$ and

$$\begin{aligned} l_q(C_8) &= l_q(N_C(C_5)) = N_F(l_q(C_5)) = N_F(Xz_1) \\ &= \{X, XXz_1, XYz_1, Xz_1z_2\}. \end{aligned}$$

Finally, since the labels $\{Yz_1, z_1z_2\}$ and $\{YYz_1, Yz_1z_2\}$ are the only ones that remain after $l_q(C_5)$ and $l_q(C_8)$ have been removed, $l_q(C_6) = \{Yz_1, z_1z_2\}$ and $l_q(C_9) = \{YYz_1, Yz_1z_2\}$. □

The proof of the next lemma is similar to the last except now the vertices in C_{10} contain two elements of X rather than just one. Let v_{ij}, $i < j$, denote $l_c^{-1}(x_ix_jz_2)$ in C_{10}. Again, note that the v_{ij}'s are those vertices whose degrees are given by Lemma 4.4.

The structure of this part of T is shown in Fig. 4. Note in Fig. 4 that there are

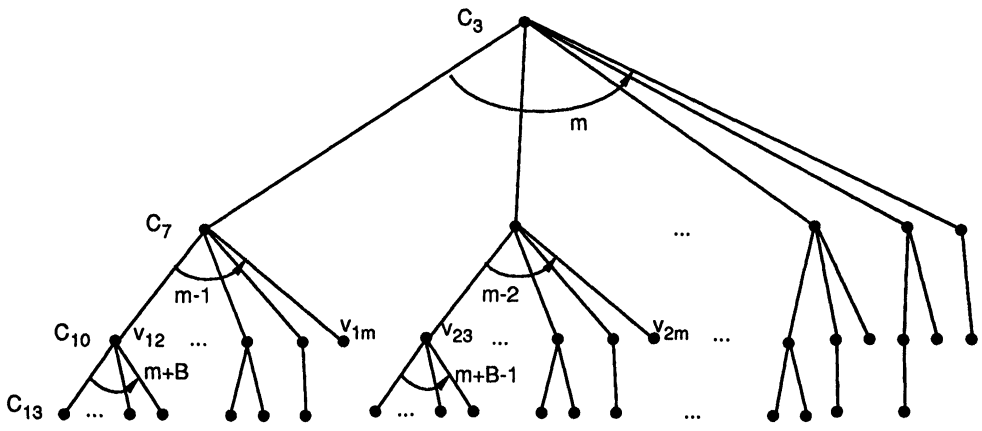


FIG. 4. *The Structure of T at* C_{10}

m subtrees rooted at C_3 and that the structure of each subtree is similar to the one occurring at C_5. The proof is similar to Lemma 6.1 except now the argument is used both within each subtree and also between subtrees. Also note that, if $C_7 = Xz_2$, then we can assume without loss of generality that the parent of v_{ij} in C_{10} is labeled x_iz_2. In addition once we have determined that $l_q(v_{ij}) \in x_iXz_2$ we can then assume

that its label is $x_i x_j z_2[j > i]$. This is a consequence of the fact that eventually *all* of the labels of the form $x_i x_j z_2[j > i]$ appear as labels in this subtree.

LEMMA 6.2. *If* $l_q(C_7) = Xz_2$, $l_q(C_{10} \cup C_{11} \cup C_{12}) = \{XXz_2, XYz_2\}$ *and* $l_q(C_{13} \cup C_{14} \cup C_{15}) = \{XX, XXXz_2, XXYz_2, XXz_1z_2, XYYz_2\}$, *then* $l_q(C_{10}) = XXz_2$, $l_q(C_{11} \cup C_{12}) = XYz_2$, $l_q(C_{14} \cup C_{15}) = XYYz_2$, *and* $l_q(C_{13}) = \{XX, XXYz_2, XXz_1z_2, XXXz_2\}$.

Proof. The proof is by induction on the vertices in C_{10}. First, lexigraphically order the vertices in C_{10} so that for any two vertices v_{kl} and v_{ij}, $kl \prec ij$ if $k < i$ or $k = i$ and $l < j$. Also, note that by definition, $i < j$ and $k < l$. Let $F = \{XX, XXXz_2, XXYz_2, XXz_1z_2, XYYz_2\}$ and let $V_{ij} = \{v_{kl} \in C_{10} \mid kl \preceq ij\}$. We claim that for all i, j the following property holds:

$$\mathrm{P}(ij): \quad l_q(v_{ij}) \in XXz_2 \text{ and } l_q(N_C(V_{ij})) = N_F(l_q(V_{ij})).$$

Again P(12) follows from the same argument given for the general case and is left to the reader. Suppose for all $kl \prec ij$ that P(kl) holds. Now, since $l_q(C_7) = Xz_2$ assume, without loss of generality, that the parent of v_{kl} is labeled $x_k z_2$. This implies that for all $kl \prec ij$, $l_q(v_{kl}) \in x_k Xz_2$. Therefore we can also assume, with no loss of generality, that $l_q(v_{kl}) = x_k x_l z_2$.

As in Lemma 6.1 we begin by determining the set of labels that can be used to label the children of v_{ij}. Let $V_{\mathrm{pred}(ij)} = \{v_{kl} \in C_{10} \mid kl \prec ij\}$, that is, the labels that, by the induction hypothesis, are known to label the children of vertices in $v_{kl}, kl \prec ij$. We have, by the inductive hypothesis, that the set of free labels

$$\begin{aligned} F' &= F - l_q(N_C(V_{\mathrm{pred}(ij)})) = F - N_F(l_q(V_{\mathrm{pred}(ij)})) \\ &= F - N_F(x_k x_l z_2[kl \prec ij]) = F - \{x_k x_l,\ x_k x_l Xz_2,\ x_k x_l Yz_2,\ x_k x_l z_1 z_2\}[kl \prec ij] \\ &= \{x_k x_l, x_k x_l x_p z_2, x_k x_l Yz_2, x_k x_l z_1 z_2, XYYz_2\}[kl \succeq ij, p > l]. \end{aligned}$$

Now, by Lemma 4.4, v_{ij} has $m + B - j + 2$ children whose label must come from F', where by hypothesis the label of v_{ij} is of the form XXz_2 or XYz_2. Without loss of generality we can assume that the parent of v_{ij} is labelled $x_i z_2$, which implies that $l_q(v_{ij})$ is a member of $x_i Xz_2$ or $x_i Yz_2$. However, if $l_q(v_{ij}) \in x_i Yz_2$ then it has only $m + B - j$ neighbours in F', $m - j + 1$ of the form $x_i x_k Yz_2[k \geq j]$ and $B - 1$ of the form $x_i YYz_2$. Thus, $l_q(v_{ij}) \in x_i x_k z_2[ik \succeq ij]$, which has exactly $m + B - j + 2$ labels; $m - j$ labels of the form $x_i x_k x_l z_2[l, k \geq j]$, B labels of the form $x_i x_k Yz_2$, 1 label of the form $x_i x_k$, and 1 label of the form $x_i x_k z_1 z_2$.

Now, since the degree of v_{ij} exactly equals the number of available labels it follows that all of these labels are used to label the children of v_{ij}, that is, $l_q(N_C(v_{ij})) = N_{F'}(l_q(v_{ij}))$. Finally, if we add these labels to the set of labels already known to be assigned to vertices in $l_q(N_C(V_{\mathrm{pred}(ij)}))$, then we have that $l_q(N_C(V_{ij})) = N_F(l_q(V_{ij}))$. Hence P($ij$) holds.

Therefore, by the principle of induction, $l_q(C_{10}) = XXz_2$ and

$$\begin{aligned} l_q(C_{13}) &= l_q(N_C(C_{10})) = N_F(l_q(C_{10})) = N_F(XXz_2) \\ &= \{XX, XXXz_2, XXYz_2, XXz_1z_2\}. \end{aligned}$$

Finally, it is also true that $l_q(C_{11} \cup C_{12}) = XYz_2$ and $l_q(C_{14} \cup C_{15}) = XYYz_2$ since these are the only labels that remain after $l_q(C_{10})$ and $l_q(C_{13})$ have been removed. □

6.2.1. The form of the skeleton of T in Q_{B+m+2}. The results from Lemma 6.1 and Lemma 6.2 are used in this section to show that the labels assigned to the skeleton of T by l_q are given by Fig. 1.

LEMMA 6.3. *If l_q is an embedding of T in Q_{B+m+2}, then*

(1) $l_q(C_1) = l_c(C_1)$, $l_q(C_2) = l_c(C_2)$, $\cdots$, $l_q(C_{18}) = l_c(C_{18})$,

(2) $l_q(C_{19}) = XY$, *and*

(3) $l_q(C_{20} \cup C_{21}) = \{X'YY,\ X'YYz_1z_2,\ X'YYYz_2\}$.

Proof. Starting at the root, consider each node in the skeleton of T.

(1) By the reflection and rotations used to obtain l_q , $l_q(C_1) = \emptyset$, $l_q(C_2) = z_1$, $l_q(C_3) = z_2$, and $l_q(C_4) = Y$.

(2) Consider the children of C_2.

(a) By the previous step, $l_q(C_2) = z_1$,

(b) the set of available labels in Q_{B+m+2}, $F = \mathcal{P}(S) - \{\emptyset, Y, z_1, z_2\}$, and

(c) $|N_C(C_2)| = |C_5 \cup C_6| = |\{Xz_1, Yz_1, z_1z_2\}| = |N_F(z_1)| = |N_F(l_q(C_2))|$.

Therefore, by label forcing, $l_q(C_5 \cup C_6) = \{Xz_1, Yz_1, z_1z_2\}$. Henceforth, we leave it to the reader to check that the conditions necessary for label forcing do hold. We simply abbreviate the previous argument to

$$l_q(C_1, C_2, C_3, C_4) \implies l_q(C_5 \cup C_6) = \{Xz_1, Yz_1, z_1z_2\},$$

which reads that if $l_q(C_1,\ C_2,\ C_3,\ C_4)$ are assigned the labels of the form outlined in the hypothesis then we can deduce, by label forcing, that $l_q(C_5 \cup C_6) = \{Xz_1, Yz_1, z_1z_2\}$.

(3) $l_q(C_1, \cdots, C_4, C_5 \cup C_6) \implies l_q(C_8 \cup C_9) = \{X, XXz_1, XYz_1, Xz_1z_2, YYz_1, Yz_1z_2\}$ Therefore, by Lemma 6.1, $l_q(C_5) = Xz_1$, $l_q(C_6) = \{Yz_1,\ z_1z_2\}$, $l_q(C_8) = \{X,\ XXz_1,\ XYz_1,\ Xz_1z_2\}$, and $l_q(C_9) = \{YYz_1,\ Yz_1z_2\}$.

(4) $l_q(C_1, \cdots, C_6, C_8, C_9) \implies l_q(C_{16}) = \{XYYz_1, XYz_1z_2, YY, YYYz_1, YYz_1z_2, Yz_2\}$.

(5) $l_q(C_1, \cdots, C_6, C_8, C_9, C_{16}) \implies l_q(C_{19}) = XY$.

(6) $l_q(C_1, \cdots, C_6, C_8, C_9, C_{16}, C_{19}) \implies l_q(C_7) = Xz_2$.

(7) $l_q(C_1, \cdots, C_9, C_{16}, C_{19}) \implies l_q(C_{10} \cup C_{11} \cup C_{12}) = \{XXz_2, XYz_2\}$.

(8) $l_q(C_1, \cdots, C_9, C_{10} \cup C_{11} \cup C_{12}, C_{16}, C_{19}) \implies l_q(C_{13} \cup C_{14} \cup C_{15}) = \{XX,\ XXXz_2, XXYz_2, XXz_1z_2, XYYz_2\}$.

Therefore, by Lemma 6.2, $l_q(C_{10}) = XXz_2$, $l_q(C_{11} \cup C_{12}) = XYz_2$, $l_q(C_{14} \cup C_{15}) = XYYz_2$, and $l_q(C_{13}) = \{XX,\ XXXz_2, XXYz_2,\ XXz_1z_2\}$. Now, to put the embedding in the form given by Fig. 1, permute the elements of X so that $l_q(C_{11}) = x_mYz_2$. Therefore $l_q(C_{12}) = X'Yz_2$, $l_q(C_{14}) = x_mYYz_2$, and $l_q(C_{15}) = X'YYz_2$.

(9) $l_q(C_1, \cdots, C_{16}, C_{19}) \implies l_q(C_{17}) = \{XXX, XXXXz_2, XXXYz_2, XXXz_1z_2, XXY, XXYYz_2, XXYz_1z_2\}$.

(10) $l_q(C_1, \cdots, C_{17}, C_{19}) \implies l_q(C_{18}) = \{x_mYY, x_mYYYz_2, x_mYYz_1z_2, YYz_2\}$.

(11) $l_q(C_1, \cdots, C_{19}) \implies l_q(C_{20} \cup C_{21}) = \{X'YY, X'YYYz_2, X'YYz_1z_2\}$.

Consequently, the form of the labels matches the forms given in the statement of the lemma. □

It was necessary to introduce the three weightless elements in the orginal definition of 3′-partition because of Steps (10) and (11) in the previous lemma. In Step (10) we showed that all labels of the form YYz_2 appear in C_{18} thus preventing them from being used to label vertices in C_{21}. This was possible only because there was a weightless part in the partition that allowed us to fix the location of the YYz_2 labels in the tree.

In the next section the fact that the form of T in Q_{B+m+2} can be determined is used to extract a solution to the 3′-partition problem from the embedding of T.

6.3. Extracting a solution to the 3′-partition problem . A solution to the 3′-partition problem can be extracted from the embedding of T in Q_{B+m+2} by examining the labels of certain vertices in T. Recall that in § 4.2 the set R was defined, where corresponding to each $r_i \in R$ there was an $a_i \in A \backslash \{a_{3m-2}, a_{3m-1}, a_{3m}\}$ such that $\deg(r_i) = s(a_i)$. In addition, $R \subseteq C_{19}$ and all together $N_C(R) = C_{20}$. Clearly, a partition of R into $m-1$, 3-element sets $\{r_{i_0}, r_{i_1}, r_{i_2}\}$ such that $\deg(r_{i_0}) + \deg(r_{i_1}) + \deg(r_{i_2}) = B$ also partitions A into $m-1$ sets $\{a_{i_0}, a_{i_1}, a_{i_2}\}$ such that $s(a_{i_0}) + s(a_{i_1}) + s(a_{i_2}) = B$.

THEOREM 6.4. *The set R can be partitioned into $m-1$, 3-element sets $\{r_{i_0}, r_{i_1}, r_{i_2}\}$ such that* $\deg(r_{i_0}) + \deg(r_{i_1}) + \deg(r_{i_2}) = B$.

Proof. Let, for $x_i \in X$,

$$R_{x_i} = \{r \in R \mid x_i \in l_q(r) \text{ for } x_i \in X\}.$$

We claim that the family of sets $\{R_{x_i}\}_{x_i \in X'}$ is a partition of R satisfying the statement of the theorem.

First, since $l_q(R) \subseteq XY$, each element of R belongs to exactly one R_{x_i}. However this may partition R into m rather than $m-1$ sets. But, recall that in defining R we excluded the three weightless elements $(a_{3m-2}, a_{3m-1}, a_{3m})$. Thus, every vertex in R has a nonzero number of children belonging to C_{20}. This implies that the vertices in R could not have received any labels of the form x_mY. Otherwise, its children would have received labels of the form x_mYY, contradicting the fact that these labels are assigned to vertices in C_{18}. Therefore, R_{x_m} is empty and the family $\{R_{x_i}\}_{x_i \in X'}$ is a partition of R into $m-1$, disjoint sets. All that remains to be shown is that the sum of the degrees of vertices within each R_{x_i} is B.

Consider R_{x_i} for some $x_i \in X'$, where $l_q(N_C(R_{x_i})) \subseteq x_iYY$. Given that $l_q(C_7) = Xz_2$, consider the subtree rooted at vertex $v_i = l_q^{-1}(x_iz_2)$ in C_7. The root of this subtree has descendants in C_{12}, C_{15}, and finally C_{21}. More specifically,

$$\begin{array}{lcl} l_q(v_i) & = & x_iz_2, \ (\text{in } C_7), \\ l_q(N_C(v_i)) & = & x_iYz_2, \ (\text{in } C_{12}), \\ l_q(N_C(N_C(v_i))) & = & x_iYYz_2, \ (\text{in } C_{15}), \quad \text{and} \\ l_q(N_C(N_C(N_C(v_i)))) & \subseteq & \{x_iYY, x_iYYYz_2, x_iYYz_1z_2\} \ (\text{in } C_{21}). \end{array}$$

Let $P_i = N_C(R_{x_i})$ and $G_i = N_C(N_C(N_C(v_i)))$. Note that P_i and G_i serve the same role here as they did in § 5.3, although in this case the conflict has already been resolved. Since $l_q(C_{20} \cup C_{21}) = \{X'YY, X'YYYz_2, X'YYz_1z_2\}$ and since G_i are the only vertices in C_{21} with labels of the form $\{x_iYY, x_iYYYz_2, x_iYYz_1z_2\}$, it follows that

$$l_q(P_i) \cup l_q(G_i) = \{x_iYY, x_iYYYz_2, x_iYYz_1z_2\}.$$

Now, the size of G_i can be established by either explicitly counting the vertices as defined in § 4.3, or by simply noting that these are the same vertices labelled in § 5.3 when we had a solution to 3′-partition. In § 5.3 the vertices in G_i were assigned all but B labels of the form $\{x_iYY, x_iYYYz_2, x_iYYz_1z_2\}$. Therefore,

$$|G_i| = |\{x_iYY, x_iYYYz_2, x_iYYz_1z_2\}| - B,$$

which implies that $|P_i| = B$. Therefore, as claimed, R_{x_i} partitions R into $m-1$ sets and within each set the sum of the degrees of the elements is B. □

This proves the two statements given in the outline of Theorem 3.1 and completes the proof that Tree-Embedding is NP-complete.

7. Concluding remarks. We have shown that the problem of deciding for a given tree T and integer k if T is embeddable in Q_k is NP-complete. Our result confirms the conjecture in [1] and [15] and greatly improves the previously known NP-completeness result for general graphs. As a result it is unlikely there are efficient algorithms for mapping applications with an irregular communication pattern onto the hypercube. In particular, this is true for trees and more generally planar graphs, both structures that arise frequently in practice.

It remains an open problem whether or not the tree T can be further restricted to trees with bounded degree. The fact that the degree of T was dependent on the size of the 3-partition problem was essential to ensure that there is a unique embedding of T in Q_{B+m+2}. It is possible to characterize, for a restricted class of binary trees, those trees that are subgraphs of a k-cube [13]. However, it is not known whether or not this can be extended to include all binary trees. There are several algorithms for embedding binary trees (with fixed dilation and expansion) [2], [21], [24] into the cube, but in general these approximations are not completely satisfactory and the problem of characterizing these trees remains.

Acknowledgments. We wish to thank the anomonyous referees for their helpful comments and suggestions.

REFERENCES

[1] F. AFRATI, C. H. PAPADIMITRIOU, AND G. PAPAGEORGIOU, *The complexity of cubical graphs*, Inform. and Control, 66 (1985), pp. 53–60.
[2] S. N. BHATT, F. R. K. CHUNG, 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 Press, 1985, pp. 274–282. Washington, DC.
[3] S. N. BHATT AND I. C. IPSEN, *How to embed trees in hypercubes*, Tech. Rep. YALEU/DCS/RR-443, Yale University, New Haven, CT, December 1985.
[4] S. BOKHARI, *On the mapping problem*, IEEE Trans. Computers, C-30 (1981), pp. 202–214.
[5] J. BONDY AND U. MURTY, *Graph Theory with Applications*, North-Holland, New York, 1976.
[6] G. CYBENKO, D. W. KRUMME, AND K. VENKATARAMAN, *Hypercube embedding is NP-complete*, in Proc. First Conference on Hypercube Multiprocessors, Society for Industrial and Applied Mathematics, Philadelphia, PA, 1986, pp. 148–157.
[7] D. DJOKOVIČ, *Distance-preserving subgraphs of hypercubes*, J. Combin. Theory Ser. B, 14 (1973), pp. 263–267.
[8] V. FIRSOV, *Isometric embedding of a graph in a boolean cube*, Kibernetika, 1 (1965), pp. 112–113.
[9] S. FOLDES, *A characterization of hypercubes*, Discrete Math., 17 (1977), pp. 155–159.
[10] M. GAREY AND R. L. GRAHAM, *On cubical graphs*, J. Combin. Theory Ser. B, 18 (1975), pp. 84–95.
[11] M. GAREY, R. L. GRAHAM, D. JOHNSON, AND D. E. KNUTH, *Complexity results for bandwidth minimization*, SIAM J. Appl. Math., 34 (1978), pp. 477–595.
[12] M. GAREY AND D. JOHNSON, *Computers and Intractability*, W.H. Freeman, San Francisco, CA, 1979.
[13] I. HAVEL, *On Hamiltonian circuits and spanning trees of hypercubes*, Časopis Pěst. Mat., 109 (1984), pp. 135–152.
[14] I. HAVEL AND P. LIEBL, *Embedding the polytomic tree into the n-cube*, Časopis Pěst. Mat., 98 (1973), pp. 307–314.
[15] I. HAVEL AND J. MORÁVEK, *B-valuations of graphs*, Czech. Math. J., 22 (1972), pp. 338–351.
[16] S. L. JOHNSSON, *Communication efficient basic linear algebra computations on hypercube architectures*, Tech. Rep. YALEU/DCS/RR-361, Yale University, New Haven, CT, November 1985.
[17] D. W. KRUMME AND K. VENKATARAMAN, *On the NP-hardness of a certain construction*, Tech. Rep. 86-1, Department of Computer Science, Tufts University, Medford, MA, May 1986.
[18] L. NEBESKÝ, *On cubes and dichotomic trees*, Časopis Pěst. Mat., 99 (1974), pp. 164–167.

[19] A. L. ROSENBERG, *Issues in the study of graph embeddings*, in Graph-Theoretic Concepts in Computer Science (Proc. Internat. Workshop WG80), H. Noltemeier, ed., Springer-Verlag, New York, 1981, pp. 151–176. Lecture Notes in Computer Science 100.

[20] J. SAXE, *Dynamic-programming algorithms for recognizing small-bandwidth graphs in polynomial time*, Tech. Rep. CMU-CS-80-102, Carnegie-Mellon University, Pittsburgh, PA, 1980.

[21] I. H. SUDBOROUGH AND B. MONIEN, *Simulating binary trees on hypercubes*, in Proc. 3rd Aegean Workshop on Computing: VLSI Algorithms and Architectures, Corfu, Greece, 1988.

[22] M. D. TYLKIN, *On Hamming geometry of unitary cubes*, Cybernetics and Control Theory, Soviet Phys. Dokl., (1961), pp. 940–943.

[23] A. WAGNER, *Embedding trees in the hypercube*, Ph.D. thesis, Tech. Report 204/87, University of Toronto, Toronto, Ontario, Canada, October 1987.

[24] ———, *Embedding arbitrary binary trees in a hypercube*, J. Parallel Distributed Comput., 7 (1989), pp. 503–520.

SIAM J. COMPUT.
Vol. 19, No. 3, p. 591, June 1990

014

ERRATUM: Weighted Leaf AVL-Trees*

VIJAY K. VAISHNAVI†

In Table 1 [1, p. 527], there should be an "*" (indicating that the corresponding result is the best known result) beside each B in the "insertion" column and beside G, N, and both D's in the "promotion" column. Reflecting this correction, the first sentence of the third paragraph [1, p. 527 (within Concluding remarks, § 6)] should read, "In terms of the worst-case time-complexities, weighted leaf AVL-trees are similar to the best solutions available [3]–[5], [8], [9]." Also, in the second to last sentence of the first paragraph [1, p. 504 (within Introduction, § 1)], the phrase "are the same or better than any data structure available in the literature" should be replaced by "match the best results available in the literature."

REFERENCE

[1] V. K. VAISHNAVI, *Weighted leaf* AVL-*trees*, SIAM J. Comput., 16 (1987), pp. 503–537.

* Received by the editors November 13, 1989; accepted for publication December 21, 1989.
† Department of Computer Information Systems, Georgia State University, Atlanta, Georgia 30303.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 593-611, August 1990

THE NUMBER OF SHORTEST PATHS ON THE SURFACE OF A POLYHEDRON*

DAVID M. MOUNT†

Abstract. It is proven that if the shortest paths on the surface of a convex polyhedron are grouped into equivalence classes according to the sequences of edges that they cross, then the resulting number of equivalence classes is $O(n^4)$, where n is the number of vertices of the polyhedron. In fact, the more general result that any family of pseudosegments (a set of open simple curves on the plane such that two curves intersect each other in at most one point) lying on a planar subdivision defined by n other pseudosegments can give rise to at most $O(n^4)$ edge sequences is also proven. This bound is shown to be asymptotically tight, by giving an example of a family of polyhedra with $\Omega(n^4)$ shortest path equivalence classes.

Key words. shortest paths, convex polyhedra, computational geometry, pseudolines

AMS(MOS) subject classifications. primary 68U05; secondary 52A25

1. Shortest path sequences. Consider a convex polyhedron P with n vertices. For a pair of points x and y on the surface of P let $\phi(x, y)$ denote a path of minimum Euclidean length from x to y along the surface of the polyhedron. Define the corresponding *shortest path edge sequence* (or simply "edge sequence") $p(x, y)$ to be the sequence of polyhedron edges that the path $\phi(x, y)$ traverses. The edge sequence can be thought of as a discrete description of the geometric path. We say that two paths on the surface of a polyhedron are *equivalent* if their edge sequences are equal. This relation partitions the set of shortest paths into equivalent classes. For example, in Fig. 1, paths ϕ_1 and ϕ_2 are equivalent but ϕ_3 is not equivalent to either ϕ_1 or ϕ_2.

For a convex polyhedron P, let $S(P)$ denote the set of edge sequences of *all* shortest paths on P. Let $s(P)$ denote the cardinality of $S(P)$ and let $s(n)$ denote the least upper bound on the value of $s(P)$ for all convex polyhedra with n vertices. In this paper we consider the combinatorial problem of bounding the value of $s(n)$. The quantity $s(n)$ was first studied by Sharir in his paper on finding shortest paths amidst

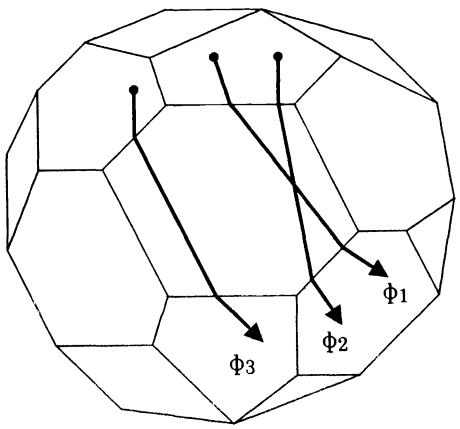


FIG. 1. *Shortest path equivalence classes.*

* Received by the editors September 14, 1987; accepted for publication (in revised form) June 2, 1988.
† Department of Computer Science and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742.

a set of convex polyhedra [5]. As is common in computational geometry, the analysis of the time and space complexity of their algorithm reduces in part to solving such a combinatorial problem. (We say "in part" because although this paper provides a bound on $s(n)$, it does not provide an algorithmic technique for generating these edge sequences.)

The quantity $s(n)$ is of interest in its own right because any attempt to preprocess a polyhedron to provide fast responses to shortest path queries may very well have to contend with this quantity in its space complexity, since this value indicates how many ways there are of getting from one place to another along the polyhedron's surface. Sharir showed that $s(n)$ is bounded above by $O(n^7)$. In this paper we provide the tight asymptotic bound of $\Theta(n^4)$ for $s(n)$. Our results have recently been extended by Schevon and O'Rourke [4] to the problem of determining the number of *maximal* edge sequences; that is, edge sequences that cannot be extended to form longer shortest path edge sequences. They provide a tight bound of $\Theta(n^3)$ maximal edge sequences.

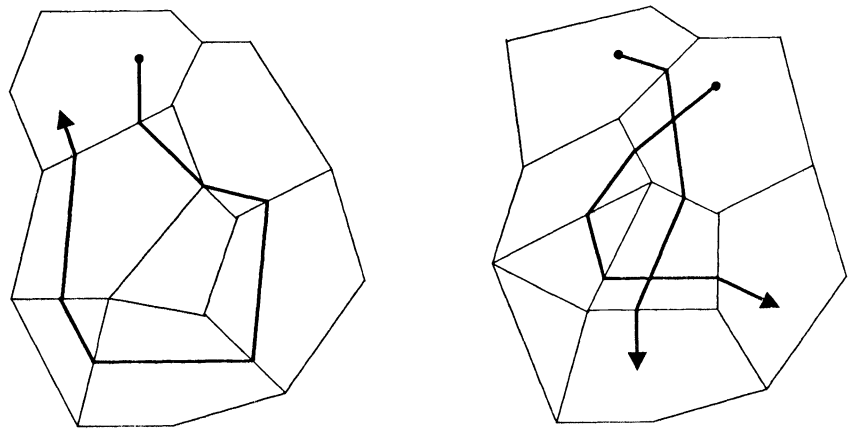
To prove the result, we map the problem to a topological domain (described in terms of embedded planar graphs) and exploit certain topological properties of shortest paths on convex polyhedra. Polyhedral shortest paths are continuous geodesic curves which intersect a given face of the polyhedron in a line segment. It will simplify the presentation if we assume that shortest paths are open curves. Some relevant topological properties of shortest paths are listed below. These follow from observations made by Sharir and Schorr [6].

(S1) Shortest paths do not pass through vertices of the polyhedron P and do not cross an edge of P more than once (see Fig. 2(a)).

(S2) No shortest path intersects itself.

(S3) Except for the case of two shortest paths sharing a common subpath, shortest paths intersect in at most one point, and this intersection is transverse, that is, the paths cross each other (see Fig. 2(b)).

Regarding property (S3), it should be mentioned that if two shortest paths share a common subpath, then either one path is a subpath of the other, or else an initial subpath of one coincides with a final subpath of the other. From property (S1) it follows that the number of shortest path edge sequences is finite. It will not affect the



(a) (b)

FIG. 2. *Invalid shortest paths.*

number of edge sequences if we consider only a finite set of representative shortest paths, one from each equivalence class. We may assume that no shortest path in the set of representatives is a subsegment of an edge of the polyhedron. Such single-edge path sequences can be generated by two points lying arbitrarily close to one another on opposite sides of the edges. We may also assume that no two representative shortest paths share a common subpath. This assumption can be met by an infinitesimal perturbation of the endpoints of the shortest paths, which will not affect their edge sequences.

It will actually simplify the proof to consider a more general setting. The vertices, edges, and faces of the polyhedron P define a graph embedded on the surface of the convex polyhedron (so that the graph's edges do not cross over one another). Let us think of P as a *subdivision* of a topological sphere, meaning simply that P is a connected planar graph embedded on a sphere (so that its edges do not cross one another) such that every face of this graph is homeomorphic to a disk (see Guibas and Stolfi [2]). Define a *family* of *pseudosegments* to be a set of Jordan arcs (simple connected arcs) on the topological sphere such that two curves share at most one point in common, at which they cross each other transversally. This is a natural counterpart to the notion of a pseudoline given by Grünbaum [1]. From properties (S1)-(S3) it follows that by homeomorphically mapping the surface of the convex polyhedron to the sphere, the edges of the polyhedron along with any finite set of shortest paths are mapped to a family of pseudosegments. Henceforth, we will forget that P is a polyhedron, and think of it as a topological subdivision of the sphere, and replace the notion of shortest path with that of a pseudosegment. The upper bound on the value of $s(n)$ follows as an easy corollary to the following theorem, which is our main result.

THEOREM 1.1. *Let P be an n-edged finite subdivision of the topological sphere, and let Σ be a finite set of open Jordan arcs on the sphere such that Σ and the edges of P form a family of pseudosegments, and the arcs of Σ do not pass through the vertices of P. Then the number of distinct edge sequences of elements of Σ that start and terminate at a given pair of edges is $O(n^2)$. Hence the total number of distinct edge sequences is $O(n^4)$.*

Although stated in terms of a subdivision of the topological sphere, the result clearly applies to planar subdivisions as well. Note that in the statement of the theorem we have changed the role of n from the number of vertices in the convex polyhedron to the number of edges of the subdivision. This distinction is unimportant when dealing with convex polyhedra because the number of edges and vertices are linearly related (by Euler's formula). However, this bound does not apply in general to planar subdivisions, which may have an arbitrary number of *multiedges* (multiple edges sharing the same endpoints) and *loops* (edges whose endpoints are equal). The allowance of multiedges and loops will be needed in our proof. In order to be a subdivision, P (as a graph) must be connected, and hence the number of vertices and faces of P is $O(n)$.

Let G denote the *planar dual graph* of P; that is, G is a graph embedded in the plane whose vertices are the faces of P and whose edges are the pairs of faces that share a common edge. Both P and G are planar graph embeddings. Let S denote the set of edge sequences of the elements of Σ with respect to P. Because we have considered pseudosegments to be open arcs, which do not travel along edges of the subdivision, each edge sequence of S can naturally be viewed as a path in the graph G (see Fig. 3). To avoid confusion, we will use the term "pseudosegment" when referring to paths on the subdivision P, and "path" when referring to paths in the dual graph G. When a pseudosegment crosses a subdivision edge between two faces of P, the corresponding path in G traverses the corresponding dual edge between these two faces in G. Our discussion will concentrate on the graph G, so unless otherwise stated, the term "edge"

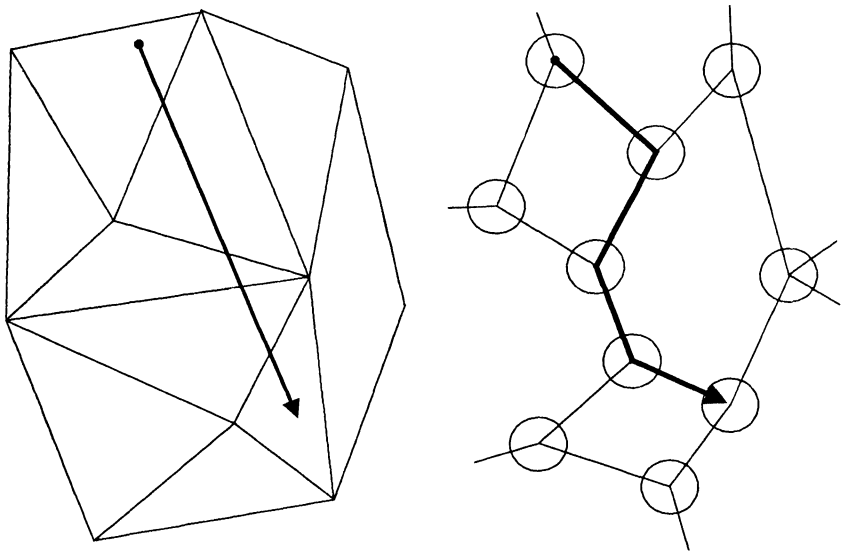


FIG. 3. *Edge sequences and the dual graph.*

will refer to an edge of G, not the subdivision P. By property (S1) no path traverses an edge of G more than once.

The paper is outlined as follows. In §§ 2 and 3 we give the proof of the upper bound. In § 4 we show that there exist convex polyhedra with $\Omega(n^4)$ edge sequences establishing the lower bound. In § 5 we consider the number of path sequences on nonconvex polyhedra.

2. Pseudosegments and paths. In the next two sections we give the proof of Theorem 1.1. The proof can be outlined as follows. In this section we introduce definitions and basic facts about paths. The principal combinatorial lemma that we prove in this section is that any set of distinct noncrossing paths emanating from a common origin in G contains $O(n)$ elements. (We define the term "noncrossing" later.) In § 3 we introduce a contraction operation that removes one edge from the graph G and in the process may cause previously distinct edge sequences to become equal. We begin with a finite set of paths in G traveling from an arbitrary source edge to an arbitrary destination edge. We show that the set of paths that become equal when a contraction is performed has essentially the same structure as a set of noncrossing paths emanating from a common origin, and hence at most $O(n)$ paths become equal with each contraction. By repeating the contraction on all n edges, we reduce the set of distinct paths by $O(n)$ each time until a trivial set of $O(n)$ paths remain. Since we have removed at most $O(n^2)$ paths in the entire process, it follows that the original number of distinct paths between the two edges was $O(n^2)$, as desired.

We begin with an introduction of the terms used throughout the proof. Consider the undirected dual graph G embedded on the sphere. Although G is undirected, it will be convenient to think of each edge as consisting of two oppositely directed edges. Each directed edge has an *origin* and *destination* vertex. For a directed edge e, let $\bar{e}$ denote its directed complement. For each vertex v, the edges whose origin is v are given in clockwise order. We use the terms *clockwise* and *counterclockwise* when referring to the exact orientation of the order, and *cyclic* when no distinction between clockwise and counterclockwise is made.

A *path* in G consists of an *origin* vertex u, a *destination* vertex v, and a possibly empty sequence of directed edges $e_1, e_2, \cdots, e_k$ connecting u to v. Given this sequence of edges, we can infer the sequence of vertices that the path visits. (The purpose of giving the origin and destination vertices explicitly is to handle the case of an empty edge sequence.) The paths that we will consider are *edge simple*, meaning that they may visit an undirected edge of G at most once. However, we allow a path to visit a vertex more than once. For this reason we distinguish between the various *instances* of a vertex along a path. (Note that shortest paths do not visit a face of the polyhedron more than once, so this allowance may seem unnecessary, but in the course of our proof we will violate this property.) Let $\bar{p}$ denote the reversal of a path p. A path q is a *subpath* of a path p if either q has an empty edge sequence and p visits the origin (and hence the destination) of q, or if q has a nonempty edge sequence that is a subsequence of p. A path q is a subpath of the *undirected* path p if q is a subpath of either p or $\bar{p}$. The path q is an *initial subpath* of p if q is a subpath starting at the origin of p. (By "origin" we mean the first instance of p's origin vertex.)

As mentioned in the introduction, the structure of a set of distinct paths emanating from a common origin will be important to the proof. The clockwise ordering of edges about each vertex of G can be extended naturally to clockwise ordering of a set of paths emanating from a common origin. Consider a path originating from a vertex v. The path can be mapped to sequences of integers as follows. First distinguish an arbitrary edge e_0 incident to v. The first integer of the sequence is the clockwise index of the first edge of the path with respect to e_0. As the path arrives at a vertex u entering along an edge e_1 and exiting along an edge e_2, the next integer in the sequence is the clockwise distance (number of edges) of e_2 from e_1 about u. Given a set of paths T originating at v, such that no path is an initial subpath of any other, a clockwise ordering of paths can be derived by ordering the integer sequences lexicographically and making the smallest numbered path the successor of the largest numbered path. This is called the *clockwise ordering of paths about* v (or *cyclic ordering* if no particular orientation is distinguished). For a list of paths $a_1, a_2, \cdots, a_k$ emanating from a common vertex v, we will let $(a_1, a_2, \cdots, a_k)$ denote the predicate that a clockwise enumeration of these paths about v starting with a_1 encounters the paths in the order $a_1, a_2, \cdots, a_k$ (equal paths may be encountered in any order).

It is almost immediate from these definitions that if a set of noncrossing pseudosegments originate from a common face of the subdivision of the topological sphere, then the resulting cyclic ordering of their edge sequences corresponds naturally to a clockwise ordering of the pseudosegments (see Fig. 4). We state this as our first lemma.

LEMMA 2.1. *Let T be a set of edge sequences corresponding to a set of pseudosegments Φ that emanate from a common face without crossing one another. If none of the edge sequences of T is a subsequence of any other, then the clockwise order of T is the same as the clockwise order of Φ.*

In order to exploit the topological property that no two shortest paths cross more than once, we will need to develop a corresponding notion of crossing between edge sequences (that is, paths in G). To begin, we say that two paths in G, p and q, *intersect* each other if, as undirected paths, they share a common subpath (possibly a single vertex). A *subpath of intersection* is any maximal common subpath ignoring path directions. A path may intersect itself, for example, if it visits some vertex v more than once.

To motivate the definition of path crossing, consider a pair of pseudosegments ϕ and ψ on the sphere. Suppose ϕ and ψ cross each other at some point x. The point x splits each of ϕ and ψ into two subsegments, ϕ_1, ϕ_2 and ψ_1, ψ_2, respectively. The

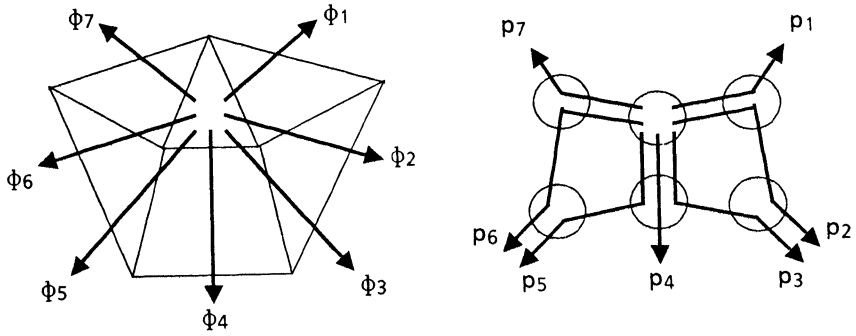


FIG. 4. *Cyclic order of paths about a vertex.*

fact that the pseudosegments cross at x implies that cyclic order of the subsegments locally about x alternates between subsegments of ϕ and subsegments of ψ, for example $(\phi_1, \psi_1, \phi_2, \psi_2)$. Now, consider two paths in G, p and q (possibly equal), that intersect each other along a subpath of intersection r. For any vertex instance on r, say v, split each path about v into two subpaths giving p_1, p_2 and q_1, q_2, respectively. Direct these four subpaths outwards from v, so that they share v as a common origin. Consider the cyclic order of these four subpaths about v. (It is easy to show that this cyclic order is independent of the choice of the vertex instance on the intersection subpath.) Three cases arise.

- If no subpath is an initial segment of another subpath, and the cyclic order of these subpaths about v alternates between the subpaths of p and the subpaths of q, we say that the paths *cross* (see p_2 and p_3 in Fig. 5).
- If no subpath is an initial segment of another subpath, and the cyclic order of the paths about v does not alternate, we say that the paths *intersect tangentially* (see p_1 and p_3 in Fig. 5).
- If one of the subpaths is an initial segment of another, then we say that the intersection is *indeterminate* (see p_1 and p_2 in Fig. 5).

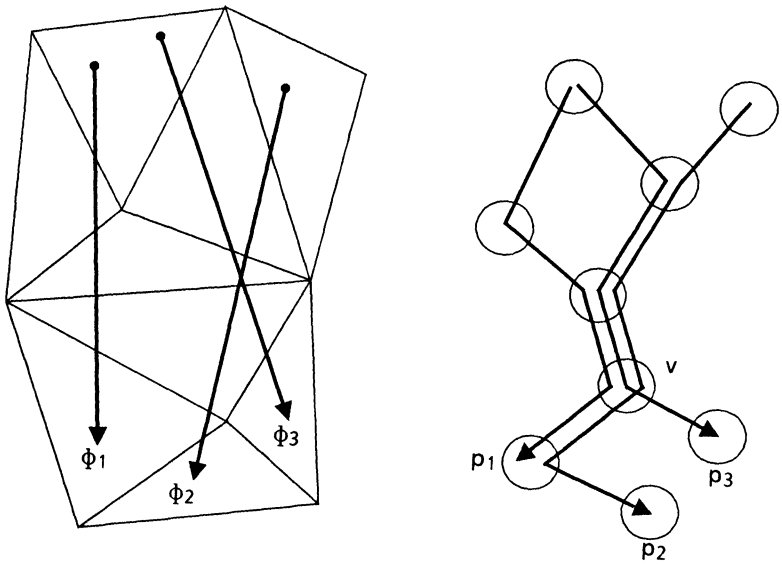


FIG. 5. *Pseudosegment and path crossing.*

The intuition behind this definition is that if two pseudosegments give rise to two crossing paths in G, then the pseudosegments must also cross, and similarly, if the pseudosegments give rise to two paths in G that intersect tangentially, then the corresponding pseudosegments must not intersect. If the path intersection is indeterminant, then the pseudosegments may or may not intersect. This happens, for example, if the pseudosegments have the same edge sequences. To formally justify this, we define the *number of crossings* between two paths of G to be the number of distinct intersection subpaths that are crossing intersections. For example, if two paths p_1 and p_2 visit a vertex k_1 and k_2 times, respectively, then p_1 and p_2 may cross each other at the vertex v as many as k_1k_2 times. We show that, like the underlying pseudosegments, two paths in G can cross at most once.

LEMMA 2.2. *Consider two paths p and q in G arising from the edge sequences of two pseudosegments ϕ and ψ, respectively. The paths p and q can cross at most once.*

Proof. We show that for each crossing intersection of p and q there is at least one corresponding crossing point x at which ϕ and ψ cross, and that this correspondence is 1-1. Because pseudosegments can cross at most once, it follows that p and q cannot cross more than once.

Suppose that p and q intersect and cross along a maximal common subpath r. Let $v_0, v_1, \cdots, v_k$ be the sequence of vertices of G visited by this common subpath. These vertices in the dual graph G correspond to a sequence of faces $f_0, f_1, \cdots, f_k$ of the subdivision P. Redirect p and q if needed so that they have the same direction along this subpath. By the maximality of the intersection subpath, and since p and q cross, it follows that p and q enter the face f_0 on different edges e_p and e_q, respectively, and depart the face f_k on different edges d_p and d_q, respectively (see Fig. 6(a)).

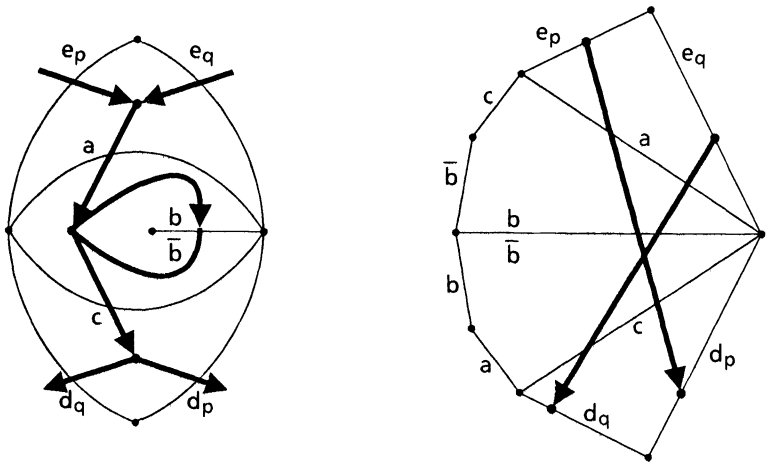


FIG. 6. *Mapping paths to curves.*

The sequence of faces $f_0, \cdots, f_k$ of P can be mapped homeomorphically into a region of the topological sphere having a simple closed boundary as follows. Since P is a subdivision, each face of P is homeomorphic to a disk. The boundary of face f_i is composed of the cyclically ordered edges incident on vertex v_i of G. (Note that a

loop about v_i is treated as two separate edges on this boundary.) Paste together the faces corresponding to the vertices $v_0, \cdots, v_k$ of the subpath r along the edges of r, and continuously deform the faces as necessary to keep them from overlapping. If the same face appears more than once in the sequence, then each instance is treated as a separate object. (For example, in Fig. 6(b) the middle face is visited twice when the edge b is traversed.) This mapping preserves the cyclic order of edges about the faces. The fact that p and q cross implies that the edges e_p and d_p alternate with the edges e_q and d_q about the boundary of this region.

The pseudosegments ϕ and ψ are mapped homeomorphically to a pair of connected curves that lie entirely within this region except at their intersection points along e_p, d_p and e_q, d_q, respectively. This mapping preserves any intersections between ϕ and ψ. Since pseudosegments do not pass through vertices of P, the paths intersect the interiors of these edges, which are mutually disjoint. The alternating order of these four edges implies that the intersection points of ϕ and ψ with the boundary of the region alternate around the boundary of the region. Thus, the two pseudosegments cross each other at some point x within the region. The crossing of ϕ and ψ at x is identified with this particular crossing intersection of p and q.

To see that this correspondence is 1-1, consider a point x at which two pseudosegments ϕ and ψ cross. This point determines a unique maximal common subpath in G between the two corresponding paths p and q. (Note that, although the edge sequence of ϕ can visit a face more than once, an occurrence of x on ϕ is identified with a unique occurrence of the face containing x in the edge sequence of ϕ.) Each maximal common subpath can give rise to at most one intersection between two paths, and hence at most one crossing intersection betwen p and q can be identified with this instance x. □

The principal combinatorial observation of this section is that the number of distinct noncrossing paths, originating from a common origin and terminating at a common destination, is linearly bounded. It is interesting to note the similarity between this and the geometric theorem that the number of distinct, maximal shortest path edge sequences that originate from a single source on a convex polyhedron is bounded by the number of vertices of the polyhedron (see Prop. 4.8 of Sharir and Schorr [6]). Although in this result, shortest paths are not necessarily maximal, the common destination edge serves to eliminate initial subpaths, which would otherwise increase the bound to $O(n^2)$.

LEMMA 2.3. *Consider a set of edge-simple paths in G, all sharing a common origin v and terminating on a common edge e, such that no two paths of the set cross each other. Then the number of paths in the set is $O(n)$.*

Proof. No path can be an initial subpath of another path because all paths terminate with the same edge, and no path can traverse this edge twice. Thus, the paths can be ordered cyclically about v. Draw the graph G on the sphere so that its edges do not cross one another, and draw the paths of the set. Since the paths do not cross, the cyclic ordering of paths partitions the sphere into a set of disjoint cyclically ordered regions about v—each consecutive pair of regions being separated by consecutive paths. Each region contains at least one face of G, for otherwise either the adjacent pairs of paths would be equal or else they would traverse some edge twice. The number of regions and hence the number of paths is bounded by the number of faces of G, which is $O(n)$. □

3. Edge contraction. Given the definitions of the previous section, we can reformulate Theorem 1.1 in terms of the paths in G. It is an immediate consequence of Lemma

2.2 and the earlier observations about the relationship between paths in G and pseudosegments that it will suffice to prove the following lemma.

LEMMA 3.1. *Consider the dual G of a subdivision P of the sphere and a set of distinct paths S in G with the following properties.*

(P1) *No path of S transverses an undirected edge of G more than once.*

(P2) *No path in S crosses itself.*

(P3) *No two paths in S cross one another more than once.*

Then the number of paths in S that start and terminate at any given pair of edges of G is $O(n^2)$. Hence the total number of paths in S is $O(n^4)$.

Although most of our proof will be given in terms of this lemma, it will be convenient to refer occasionally to the underlying set of pseudosegments Σ that gave rise to S. Our analysis works by iterating a topological transformation that maps G into successively smaller graphs. The intuition behind this transformation can be seen by recalling that G is the dual of a subdivision P. One method to simplify P iteratively is to remove each of its edges. To maintain the property that P is a subdivision, we must take care when removing edges that the boundary of each face of P remains homeomorphic to a disc, implying that its boundary must be simply connected [2]. Consider the deletion of an e of P and the subsequent removal of any isolated vertices, resulting in a subdivision P' of the sphere. If the edge lies between two distinct faces, this operation has the effect of merging the faces of P lying to either side of e into a single face in P'. (In Fig. 7(a) the subdivision is shown with light lines, and the dual graph G is shown in heavy lines.) If the edge e is incident with the same face on both sides (implying that e is a loop edge in the dual graph), then the edge can be removed, provided that one of the vertices of e is incident only to e (Fig. 7(b)). Otherwise the deletion of e would result in breaking the boundary of a face into two components (Fig. 7(c)).

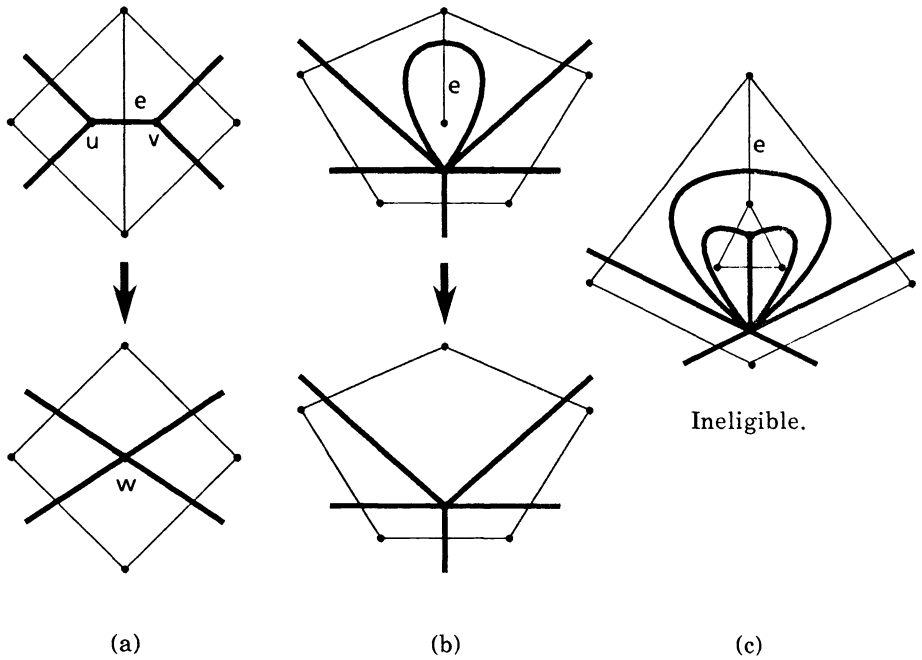


FIG. 7. *Edge contraction.*

Pseudosegments are not affected by this operation, but the edge e will be deleted from every edge sequence in which it appears. As a consequence, pseudosegments previously having different edge sequences in P may now have the same edge sequence in P'. Hence the number of distinct edge sequences may decrease. The analysis centers on determining how large a decrease may occur with each such transformation.

Let G' denote the dual of P'. G' can be defined directly in terms of G, without considering the subdivisions P and P'. The corresponding action on the dual graph G is called an *edge contraction.* Intuitively, this involves shrinking an edge of G down to a single point and merging the endpoints of this edge into a single vertex (see Fig. 7(a)). The case that the edge in G is a loop (implying that the corresponding edge in P has the same face on either side) requires special consideration. We say that a loop edge e on a vertex v is *empty* if e and $\bar{e}$ are consecutive in cyclic order about v (Fig. 7(b)). We declare nonempty loop edges to be *ineligible* for contraction. Let e be a nonloop edge or empty loop edge in G. The *contraction* of e in G is an embedded graph G' defined by the following rules.

(1) If e is a nonloop edge between endpoints u and v, form G' by deleting both e and $\bar{e}$ and by replacing u and v with a new vertex w. The edges of w are a cyclic concatenation of the remaining edges about u and v joined together at the positions that e and $\bar{e}$ occupied respectively. See Fig. 7(a).

(2) If e is an empty loop edge, then form G' by deleting e from G. See Fig. 7(b).

The facts that G' is the dual of P' and that P' is a subdivision of the sphere are easily verified. An edge contraction induces a transformation on a set of paths in G. A path p in G is mapped to a path p' in G' by deleting the edge e or $\bar{e}$ if it occurs in p. It is easy to see that property (P1) will be preserved in the new set of paths. It can be proved (somewhat arduously) by purely graph theoretic means that the number of path crossings does not increase after a contraction is performed. However, it will suffice for our purposes to show that (P2) and (P3) are preserved by appealing to the underlying pseudosegments. Since the pseudosegments are unchanged, they satisfy properties (P2) and (P3). Since Lemma 2.2 assumes that P is any subdivision of a topological sphere, we can apply this lemma to P' and its dual G'. Properties (P2) and (P3) follow immediately. Thus we have

LEMMA 3.2. *Properties* (P1)-(P3) *are preserved under edge contraction.*

Returning to the proof of Lemma 3.1, we wish to show that the number of paths in S between any pair of edges, say, e_s and e_t is $O(n^2)$. Let $S(e_s, e_t)$ denote the set of paths of S that start with e_s and terminate with e_t. Let $\Sigma(e_s, e_t)$ denote the corresponding finite subset of representative pseudosegments that gave rise to these paths. If $e_s = e_t$, then $S(e_s, e_t)$ consists at most of the single edge sequence e_s, because pseudosegments can cross an edge at most once. Thus we confine our attention to the more interesting case in which e_s and e_t are distinct.

Recall that edge contraction can be performed only on nonloop and empty loop edges of G. At some point no more contractions can be performed because all remaining edges are nonempty loops. To simplify the analysis at this limiting stage, we augment the graphs P and G by adding a special pseudosegment between e_s and e_t. Assuming that $\Sigma(e_s, e_t)$ is nonempty, let $\phi(e_s, e_t)$ be an arbitrary pseudosegment from this set. Because there are finitely many pseudosegments in $\Sigma(e_s, e_t)$, by a slight perturbation of $\phi(e_s, e_t)$, we may assume that $\phi(e_s, e_t)$ does not cross any edge of the subdivision P at the same point that any other pseudosegment crosses this subdivision edge. Let $W = (w_1, w_2, \cdots, w_k)$ denote the sequence of intersections of $\phi(e_s, e_t)$ with the edges of P. Augment P by adding new vertices W to P (thus splitting each of these edges of the subdivision into two edges) and adding undirected edges (w_i, w_{i+1}), for $1 \leqq i < k$.

Each of the newly added subdivision edges of P is a subsegment of $\phi(e_s, e_t)$ (see Fig. 8). Augment the dual graph G accordingly.

Note that the graph P is still a subdivision of the sphere. Further, since $\phi(e_s, e_t)$ can cross each edge of P at most once, the number of subdivision edges is at most three times the original number of edges in P (one new edge for each (w_i, w_{i+1}) and one new edge for each edge of P split by a vertex w_i). Remove $\phi(e_s, e_t)$ from $\Sigma(e_s, e_t)$, and remove the edge sequence for $\phi(e_s, e_t)$ from $S(e_s, e_t)$. Augment the remaining elements of $S(e_s, e_t)$ if they cross any of the edges (w_i, w_{i+1}). It is easy to verify that the conditions of Theorem 1.1 and the conditions of Lemma 3.1 are still satisfied.

Let D denote the set of edges consisting of e_s, e_t (which have now been split into four distinct edges) and the edges (w_i, w_{i+1}). Let us redefine n to be the number of edges in P excluding the edges of D. The new n is no more than twice the original value, so asymptotically the results are not altered. We construct a sequence of graphs $G_n(=G)$, $G_{n-1}, \cdots, G_k$ by successively contracting any eligible nonloop edges or empty loops with the exception of the edges of D. Let P_i denote the corresponding planar dual graph of G_i. The sequence is terminated when there are no more edges eligible to be contracted. We begin by showing that, other than the expected edges, every edge in G can be contracted.

LEMMA 3.3. *If G_k contains no edge eligible for contraction, then G_k consists only of the edges of D (implying $k=0$).*

Proof. The edges of G_k and its dual P_k are in 1-1 correspondence, so we may consider these same edges in P_k. The hypothesis of the lemma implies that the deletion of any edge $e \notin D$ (and the removal of isolated vertices) in P_k would result in a graph that has a face with a disconnected boundary. A planar graph has a face with a disconnected boundary if and only if the graph is itself disconnected. P_k is planar, and hence any graph formed by edge deletion is also planar. Since P_k is connected, this implies that each edge $e \notin D$ is a bridge. Because the edges of D form a tree in P_k, we see that P_k must be cycle free (and thus a tree), for otherwise it would contain an edge outside of D whose removal would not disconnect P_k. In particular, if P_k contains edges besides those of D, it must contain an edge $e_0 \notin D$ incident to a leaf of P_k. However, the deletion of e_0 (together with the removal of the leaf) will not disconnect P_k. Thus, $e_0 \notin D$, but e_0 is eligible for contraction, a contradiction. □

Recall that with each edge contraction we form an updated set of paths in the contracted graph. Let $S_n(=S(e_s, e_t))$, $S_{n-1}, \cdots, S_0$ denote the sets of paths running between e_s and e_t in $G_n, G_{n-1}, \cdots, G_0$, respectively. To prove that $|S(e_s, e_t)|$ is $O(n^2)$ it will suffice to establish two bounds:

- $|S_0|$ is $O(n)$, and
- $|S_k| - |S_{k-1}|$ is $O(n)$ for $1 \leqq k \leqq n$.

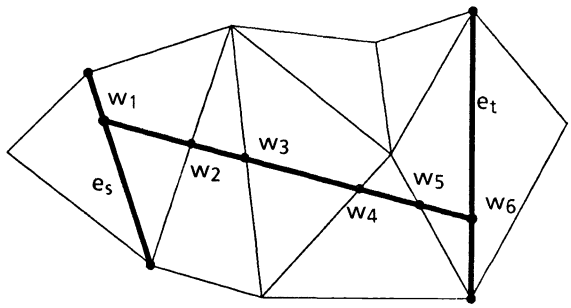


FIG. 8. *Augmenting the subdivision.*

The first bound is easy to see. G_0 and P_0 consist only of the edges of D by Lemma 3.3. The edges of P_0 consist of the edges e_s and e_t with a simple path p (the edge sequence of a pseudosegment) running between them. Because the path p is derived from a pseudosegment, any path in $S(e_s, e_t)$ can cross this segment at most once. An edge sequence of S_0 either crosses none of the edges of p, or it crosses exactly one edge in one of two possible directions. Thus the number of distinct edge sequences in S_0 is $O(|D|)$. The number of edges in D is no greater than the original number of edges in P. Therefore, the number of distinct edge sequences in S_0 is $O(n)$.

The remainder of the section is devoted to proving the second bound. The presentation is broken roughly into two parts. First, we provide a series of lemmas characterizing the structure of paths. These culminate in Lemma 3.9, which relates $|S_k| - |S_{k-1}|$ to the size of a set of paths that intersect nontangentially at a single vertex. Next, we give a bound on the size of this set, which results from a monotonicity property of its paths.

As an edge is contracted in G_k to form G_{k-1}, the paths in S_k that traverse the contracted edge e are mapped into paths in S_{k-1} by deleting the edge e. In the process, two distinct paths p and p' in S_k may be mapped onto the same path after the deletion of e. We say that such paths in S_k are *equivalent modulo e*, or simply *equivalent*. The set of equivalence classes modulo e has a very particular structure. We begin by presenting some observations on the structure of paths forming these equivalence classes. In the following lemmas, let S denote any of the sets of paths S_k for some k, $1 \leqq k \leqq n$.

LEMMA 3.4. *If two distinct paths in S share common starting and terminating edges and are equivalent modulo an edge e, then e is a loop.*

Proof. Let p and p' be distinct paths that become equal after contracting e. Let u and v be the endpoints of e. One of the paths, say, p', traverses e from u to v. The other path p traverses exactly the same sequence of vertices as p' up to u and after v. Thus p travels from u to v without the use of e (or any other edge), implying that $u = v$. □

LEMMA 3.5. *Let p be a path in S that passes through a vertex v by entering along an edge a (so that $\bar{a}$ has origin v), traversing an empty loop edge e, and then leaving along an edge b. The direction in which p traverses e is determined by the relative orientation of $\bar{a}$, b, and e about v.*

Proof. By property (P1) all edges are distinct. If the edges appear in the cyclic order $(\bar{a}, e, \bar{e}, b)$, then p traverses e in its natural direction, for otherwise, p would cross itself at v, violating property (P2). If they appear in the cyclic order $(\bar{a}, \bar{e}, e, b)$, then p traverses $\bar{e}$ by a symmetric argument. □

LEMMA 3.6. *Let p and q be two paths in S that intersect at (some instance of) a vertex v, where v is incident to an empty loop e. Suppose that paths p' and q', formed by inserting e into p and q, respectively, at this instance of v, are also in S. Then the paths p and q cannot intersect tangentially at v.*

Proof. Without loss of generality we assume that e is the clockwise successor of $\bar{e}$ at v since the actual direction in which p and q traverse e is determined by Lemma 3.5. Suppose that p' and q' intersect tangentially at v. This means that the paths p and q can be split about v into subpaths p_1, p_2 and q_1, q_2, respectively (directed outwards from v), such that they appear as (p_1, p_2, q_1, q_2) in clockwise order about v. There are four positions in which the (empty) loop e may appear in the clockwise order.

The first case is that e falls between p_1 and p_2 in cyclic order. Then path q' crosses the path p twice, once by considering the alternating cyclic order of p_1 and p_2 (of p)

with q_1 and e (of q') and the other by considering the alternating order of p_1 and p_2 (of p) with $\bar{e}$ and q_2 (of q') (see Fig. 9(a)).

Second, if e falls between p_2 and q_1, then we claim that p' crosses q' twice. One crossing is evidenced by the alternating cyclic order of p_1 and e (of p') with $\bar{e}$ and q_2 (of q') (see Fig. 9(b)). To see the other crossing, first note that both paths visit v twice. By Lemma 3.5 the paths $p' = \overline{p_1} e p_2$ and $q' = \overline{q_1} e q_2$ traverse e in the same counterclockwise direction. (Fig. 9(c) shows the two instances of v explicitly.) By hypothesis (p_1, e, q_1) is a clockwise ordering of these three paths about v as is $(p_2, \bar{e}, q_2)$. These clockwise orientations imply that the two paths $\bar{e}p_1$ and p_2 of p' alternate about the second instance of v with the paths $\bar{e}q_1$ and q_2 of q', giving us the second crossing.

If e falls between q_1 and q_2, the situation is entirely symmetric to the first case, and if e falls between q_2 and p_1, the situation is symmetric to the second case. In all cases we have shown that the property (P3) is not satisfied, a contradiction. □

LEMMA 3.7. *Let p, p' and q, q' be paths satisfying the conditions of Lemma* 3.6. *One of paths p or p' crosses one of q or q'.*

Proof. By Lemma 3.6 it follows that p and q do not intersect tangentially. If p and q cross at v, then we are done. Therefore, we may assume that p and q neither intersect tangentially nor do they cross. This implies that if we split these two paths about v into subpaths p_1, p_2 and q_1, q_2, respectively, (directed outwards from v), then at least one of these subpaths is an initial subpath of another. We may assume without loss of generality that p_1 and q_1 terminate at the edge e_s, and p_2 and q_2 terminate at e_t. This implies that either $p_1 = q_1$ or $p_2 = q_2$ (but not both, for otherwise $p = q$). Let us assume the former. The latter case follows by a symmetrical argument.

Without loss of generality we may assume that $(p_1 = q_1, p_2, q_2)$ is a clockwise ordering of these paths about v, since the other ordering follows by switching the roles of p and q. We may also assume that e is the clockwise successor of $\bar{e}$ about v, since

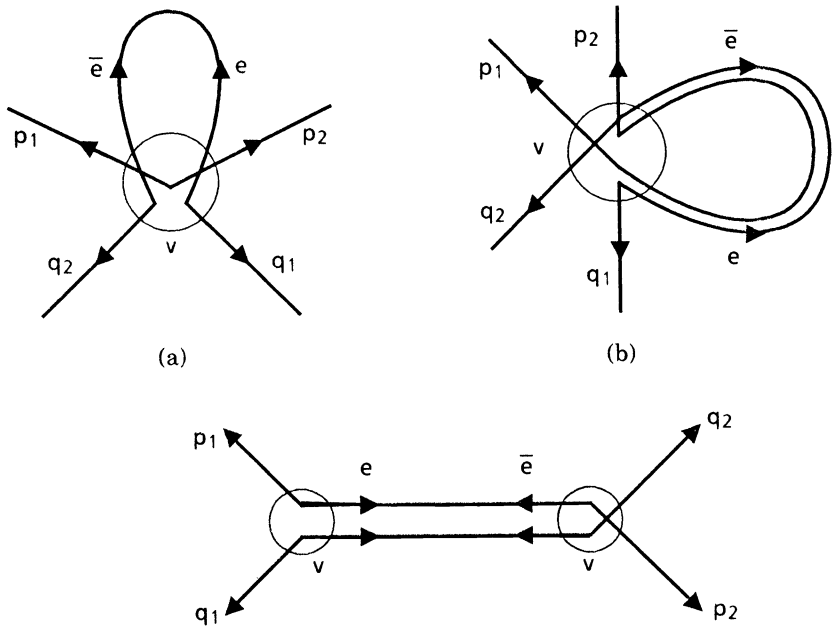


FIG. 9. *Proof of Lemma* 3.6.

the actual direction in which p and q traverse e is determined by Lemma 3.5. The loop e may fall in any of three places between these paths. If e falls between p_1 and p_2, then the paths cross by considering the alternating cyclic order of p_1 and p_2 (of p) with e and q_2 (of q') (see Fig. 10(a)). Symmetrically, if e falls between q_2 and p_1, then the paths cross by considering the alternating cyclic order of q_1 and q_2 (of q) with $\bar{e}$ and p_2 (of p').

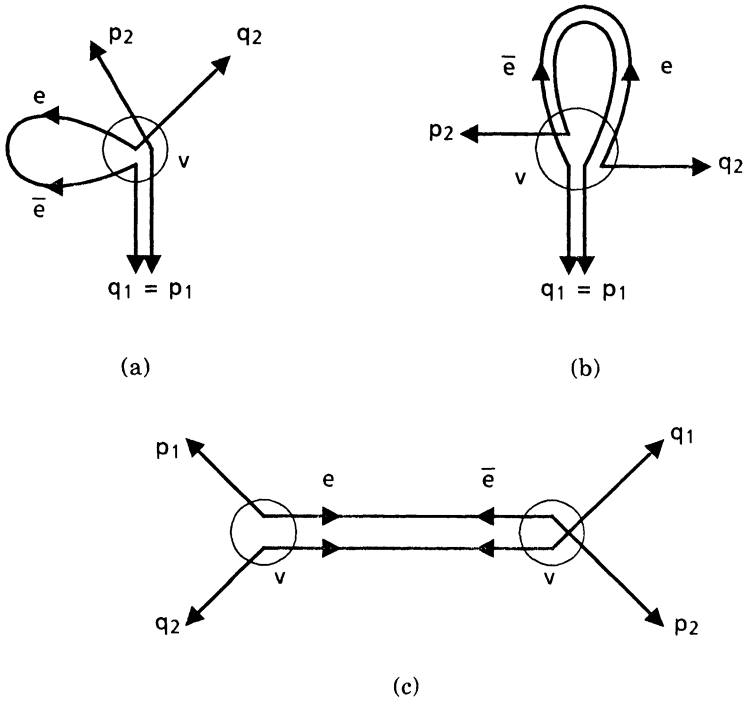


FIG. 10. *Proof of Lemma* 3.7.

Finally, if e falls between p_2 and q_2 (Fig. 10(b)), then first note that both paths visit v twice. By Lemma 3.5 the paths $p' = \overline{p_1} e p_2$ and $\overline{q'} = \overline{q_2} e q_1$ traverse e in the same counterclockwise direction. (Fig. 10(c) shows the two instances of v explicitly.) By hypothesis (p_1, e, q_2) is a clockwise ordering of these three paths about v as is $(p_2, \bar{e}, q_1)$. The clockwise orientations imply that the two paths $\bar{e}p_1$ and p_2 of p' alternate about the second instance of v with the paths $\bar{e}q_2$ and q_1 of $\overline{q'}$. This implies that p' and q' cross, completing the proof. □

LEMMA 3.8. *If two distinct paths in S are equivalent modulo e (where e is distinct from e_s and e_t), then one path differs from the other only in that one traverses the (undirected) edge e and the other does not.*

Proof. By Lemma 3.4 we may assume that e is a loop about some vertex v. By property (P1), each path traverses e at most once. Suppose to the contrary that both paths traverse e (in possibly different directions). These two different strings arise by taking a common path r that visits the vertex v at least twice, inserting the loop e or $\bar{e}$ into one instance of v on r to form p, and inserting e or $\bar{e}$ into another instance of v on r to form q. This implies that the two paths each can be decomposed into three subpaths p_1, p_2, and p_3, where p_1 and p_3 are directed outwards from v, such that one of the paths can be written $p = \overline{p_1} e p_2 p_3$ and the other path can be written

$q = \overline{p_1} p_2 e p_3$. (We are treating e as an undirected edge here. The actual direction in which q and p traverse e is determined by Lemma 3.5.) Because the paths do not start or end at e, the subpaths p_1 and p_3 are nonempty. The subpath p_2 is nonempty, for otherwise p and q would not be distinct.

The paths p_1, p_2, $\overline{p_2}$, and p_3 all share v as an origin and, by property (P1), none is an initial subpath of any other, so we may consider the cyclic order of these paths about v. There are essentially three ways that these four paths may be cyclically ordered about v (allowing for clockwise-counterclockwise symmetry). We show that all three orders lead to a contradiction.

If $(p_1, \overline{p_2}, p_2, p_3)$ is the cyclic order of the paths about v (see Fig. 11(a)), then the alternating order of p_1, p_2 and $\overline{p_2}$, p_3 implies that, regardless of the position of e, the path p contracted by the edge e crosses over itself. This violates condition (P2).

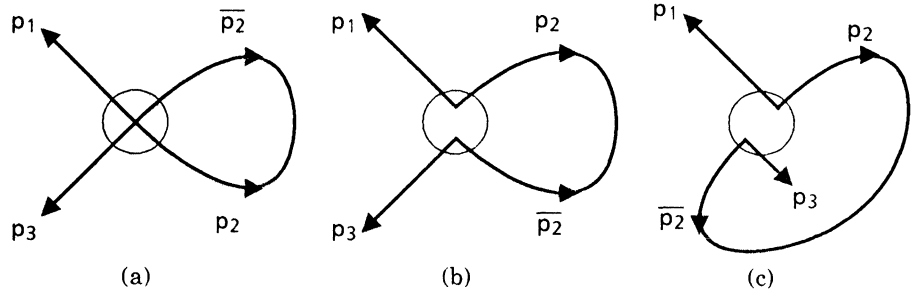


FIG. 11. *Proof of Lemma* 3.8.

If either $(p_1, p_2, \overline{p_2}, p_3)$ or $(p_1, p_2, p_3, \overline{p_2})$ is the cyclic order of the paths about v (see Fig. 11(b) and 11(c)), then the four subpaths $\overline{p_1} p_2$ (of q), $\overline{p_1} e p_2$ (of p), $p_2 p_3$ (of p), and $p_2 e p_3$ (of q) satisfy the conditions of Lemma 3.6 (as p, p', q, and q', respectively, in the statement of the lemma). However, in both cases p and q intersect tangentially at v, contradicting Lemma 3.6. □

By combining these lemmas we have a characterization of the paths that are equivalent modulo e in terms of a set of paths that intersect nontangentially at a single vertex. This lemma is illustrated in Fig. 12.

LEMMA 3.9. *For an arbitrary k, $1 \leq k \leq n$, let e be the edge contracted in forming G_{k-1} from G_k.*

(i) *$|S_k| - |S_{k-1}|$ is at most the number of equivalence classes modulo e, and this value is nonzero only if e is a loop.*

(ii) *If e is a loop incident on a vertex v, then the set of equivalence classes modulo e in S_k is in 1-1 correspondence with a set of paths T that satisfy the following properties.*

(T1) *Every path in T starts at the edge e_s and terminates at the edge e_t.*

(T2) *No two paths in T intersect each other tangentially at vertex v.*

(T3) *No two paths in T cross each other except possibly at v.*

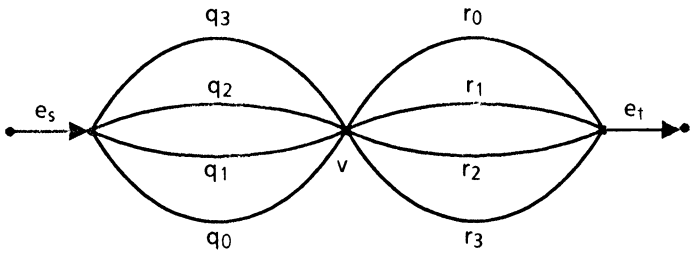


FIG. 12. *Paths equivalent modulo.*

Proof. In order to prove (i), first note that by Lemma 3.8 each equivalence class modulo e is of size one or two. This means that the difference in size between S_k and S_{k-1} is at most the number of equivalence classes modulo e, since each such class can lose at most one path in forming S_{k-1}. Let $m = |S_k| - |S_{k-1}|$. It follows from Lemma 3.4 that m is nonzero only if e is a loop. To prove (ii), let $\{(p_i, p_i') \mid 1 \leqq i \leqq m\}$ be the set of equivalence classes of size two. By Lemma 3.8 we may assume that p_i and p_i' differ only in that one, say p_i', traverses the loop e and the other, p_i, does not. Let T be the set of paths $\{p_i \mid 1 \leqq i \leqq m\}$. Clearly the paths of T satisfy property (T1).

Consider an arbitrary pair of equivalence classes (p_i, p_i') and (p_j, p_j'). Because all paths start at edge e_s and terminate at edge e_t, neither of which equals e, we can apply Lemmas 3.6 and 3.7 (where p_i, p_i', p_j, p_j' take the place of p, p', q, q', respectively, in the statements of the lemmas). Lemma 3.6 implies that the paths of T satisfy property (T2). Lemma 3.7 implies that for every pair of equivalence classes, one path from one class crosses some other path of the other class, and all of these crossings occur at the vertex v. Suppose, for example that p_i and p_j' cross each other at e. The two paths of p_i and p_j cannot cross one another elsewhere, for if they did, p_j' must also cross p_i, since p_j' is identical to p_j everywhere other than at e. This would imply that p_i and p_j' cross each other twice, violating property (P3) of paths. Therefore p_i and p_j do not cross each other except at v, establishing property (T3). □

We have reduced the problem to bounding the size of the set T. Each path $t_i \in T$ can be split about its instance of v (the instance at which the loop edge e is contracted) into two subpaths, directed outwards from v, q_i, and r_i, such that $t_i = \overline{q_i} r_i$. Consider the set of paths $Q = \{q_i \mid 1 \leqq i \leqq m\}$ and $R = \{r_i \mid 1 \leqq i \leqq m\}$. Every path in Q originates at v and terminates at e_s, and every path in R originates at v and terminates at e_t. By property (T3), no two paths in Q cross each other, and no two paths in R cross each other. Moreover, no path in q crosses a path in R.

LEMMA 3.10. *The paths of $Q \cup R$ can be ordered clockwise about v. Both Q and R contain $O(n)$ paths.*

Proof. To prove the first claim, it suffices to show that no path of Q or R is an initial subpath of any other path in $Q \cup R$. No path in Q can be an initial subpath of any other path in Q because every path in Q terminates at e_s, and no path can traverse an edge twice. No path in Q can be an initial subpath of a path in R, for if any path of R traverses e_s, then the corresponding path in T traverses e_s twice. A symmetric argument shows that no path in R is an initial subpath of either R or Q, completing the proof. The second claim follows from Lemma 2.3 since the paths of Q and R do not cross each other. □

Consider the clockwise order of $Q \cup R$ about v. We can convert the clockwise order into a linear order by distinguishing a path $q_0 \in Q \cup R$ and, for $q_1, q_2 \in Q \cup R$, defining $q_1 < q_2$ if q_1 comes before q_2 in a clockwise enumeration of $Q \cup R$ starting at q_0. Let us call this the *linear order induced by* q_0. The elements of T may be viewed as elements of the set $Q \times R$. Fix an element $p_0 \in T$ and let q_0 and r_0 be the *components* of p_0 in Q and R, respectively. We now show that the elements of T possess a monotonicity property with respect to the linear orders on Q and R as induced by q_0 and r_0, respectively.

LEMMA 3.11. *Let $p_1 = q_1 r_1$ and $p_2 = q_2 r_2$ be two elements of T, where q_1 and q_2 are distinct from q_0, and r_1 and r_2 are distinct from r_0. If $q_1 < q_2$ in the order induced by q_0, then $r_1 \leqq r_2$ in the order induced by r_0 (see Fig. 12).*

Proof. Because all of the elements of T intersect nontangentially at v, any cyclic enumeration encounters the components of p_0, p_1, and p_2 in an alternating fashion

(where some of the components may be equal). The possible cyclic orders of q_0, q_2, r_0, and r_2 are (q_0, q_2, r_0, r_2) and (q_0, r_2, r_0, q_2). In the first case, suppose that $q_1 < q_2$ in the order induced by q_0, that is, (q_0, q_1, q_2). Then, by the alternating order of components, the cyclic order of the components is $(q_0, q_1, q_2, r_0, r_1, r_2)$. This implies that $r_1 \leqq r_2$ in the order induced by r_0. We prove the lemma in the second case by contradiction by supposing that $r_1 > r_2$ in the order induced by r_0, that is, (r_0, r_2, r_1). By the alternating order of components this implies that $(r_0, r_2, r_1, q_0, q_2, q_1)$, and hence $q_2 \leqq q_1$ in the order induced by q_0. □

LEMMA 3.12. *T contains $O(n)$ paths.*

Proof. Recall the fixed element $p_0 = q_0 r_0$ chosen earlier. We split T into three subsets: T_1 contains the elements of T of the form $q_0 r$, for $r \in R$, T_2 contains the elements of T of the form $q r_0$, for $q \in Q$, and T_3 contains the elements of T that contain neither q_0 nor r_0. Clearly, every element of T is in at least one of these sets. The size of T_1 is at most $|R|$. The size of T_2 is at most $|Q|$. Thus, both sets are of size $O(n)$ by Lemma 3.10. The elements of T_3 satisfy the monotonicity condition of Lemma 3.11. Viewed as ordered pairs from $Q \times R$, if these elements are arranged in lexicographic order they are nondecreasing in their second component. Thus, each element differs from its predecessor by an increase in either the first or second component. The elements can increase at most $|Q|-1$ times in their first component and at most $|R|-1$ times in their second component. Therefore the number of elements in T_3 is at most $(|Q|-1)+(|R|-1)+1$, which by Lemma 3.10 is $O(n)$. □

Summarizing the previous discussion, we have shown that the set $S(e_s, e_t)$, consisting of the set of edge sequences starting at an edge e_s and terminating at edge e_t, is decomposed by edge contractions into a sequence of path sets, $S_n (= S(e_s, e_t))$, $S_{n-1}, \cdots, S_0$. Clearly,

$$|S_n| = \sum_{1 \leqq k \leqq n} (|S_k| - |S_{k-1}|) + |S_0|.$$

We showed earlier that $|S_0|$ is $O(n)$; thus it suffices to show that $|S_k| - |S_{k-1}|$ is $O(n)$. In Lemma 3.9 we showed that for each k, $|S_k| - |S_{k-1}|$ is equal to the size of a set of paths T, and in Lemma 3.12 we showed that T contains $O(n)$ paths. Thus, $S(e_s, e_t)$ contains $O(n^2)$ paths. Lemma 3.1 and Theorem 1.1 follow immediately.

4. A lower bound. We show that for each positive integer n, there is a convex polyhedron with $O(n)$ vertices and $\Omega(n^4)$ distinct shortest path edge sequences. Given n, consider a subdivision P consisting first of a rectangle R containing n vertical line segments, $v_1, \cdots, v_n$, each of height equal to the height of R (the center of Fig. 13). Extend the diagonals of R, and consider n rectangles concentric with R with vertices

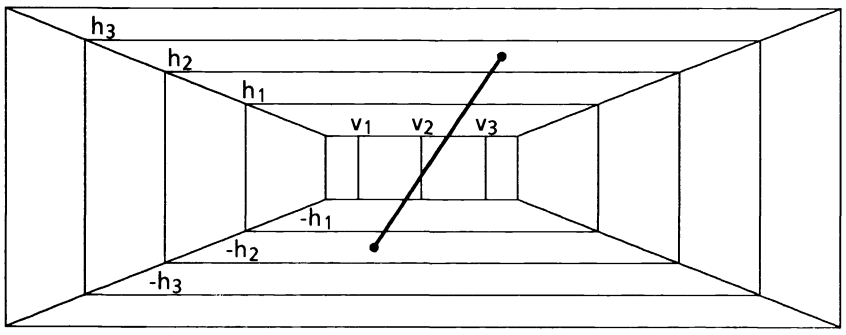


FIG. 13. *Subdivision with $\Omega(n^4)$ edge sequences.*

along the extended diagonals (the outer part of Fig. 13). Let $h_1, \cdots, h_n$ denote the horizontal edges of these rectangles above R and let $-h_1, \cdots, -h_n$ denote the horizontal edges below R.

For each pair (i, j), $1 \leqq i, j \leqq n$, there is a line that enters the top edge of R and crosses all of the vertical lines from v_i to v_j and exits the bottom edge of R. Any such line crosses all of the edges h_k and $-h_k$, $1 \leqq k \leqq n$. For any such line and any pair (k, l), consider the segment of the line that starts just above h_k and ends just below $-h_l$. The edge sequence of this segment consists of $h_k, h_{k-1}, \cdots, h_1$, followed by the vertical segments from v_i to v_j, followed by the segments $-h_1, -h_2, \cdots, -h_l$. Thus, each quadruple (i, j, k, l) generates a line segment with a different edge sequence, implying that there are $\Omega(n^4)$ different edge sequences on P.

The conclusion follows by projecting the vertices of P onto a hemisphere of sufficiently large radius centered above the middle of R, thus warping the subdivision slightly. It is easy to see that the edges of the subdivision will be mapped to edges of the convex hull of projected vertices. As the radius of the hemisphere increases, geodesics on the convex hull approach line segments on the subdivision. Hence, for all sufficiently large radii, the edge sequences given above are edge sequences of shortest paths on the convex hull.

5. Edge sequences on nonconvex polyhedra. The essential difference between shortest paths on the surface of a nonconvex polyhedron and shortest paths on the surface of a convex polyhedron is that in the nonconvex case shortest paths may pass through vertices. The analogue to edge sequences are sequences of edges and vertices. When shortest paths pass through vertices, we can construct degenerate cases in which there are exponentially many paths. To see this, let $P(x, y)$ denote the pyramid whose base is the unit square with opposing corners at $(x, y, 0)$ and $(x+1, y+1, 0)$ and whose apex is above the center of the square at height 1, and let $C(x, y)$ denote the unbounded cylinder whose base is the same unit square as $P(x, y)$ that extends downwards for $z \leqq 0$. Let $P_n(z)$ be the unbounded nonconvex polyhedron $\bigcup_{i=0}^{n} P(i, i) \cup C(i, i)$ (see Fig. 14).

It is easy to see that any shortest path from $(0, 0, 0)$ to $(n, n, 0)$ does not pass below the x, y-plane, and hence passes through the $n-1$ intermediate points $(i, i, 0)$,

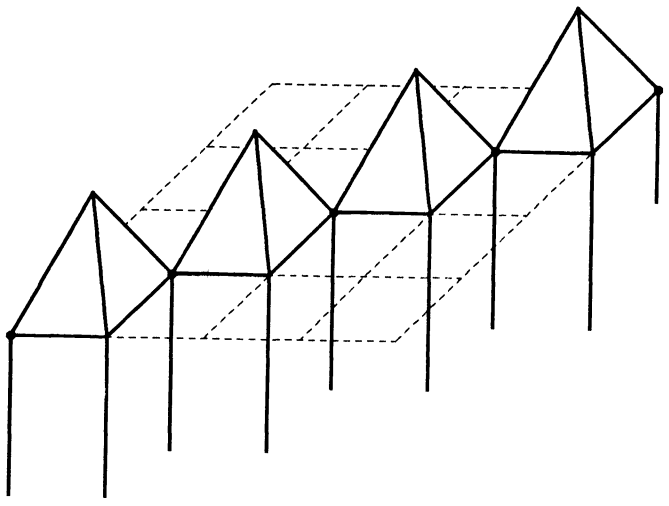


FIG. 14. *Degenerate edge sequences on nonconvex polyhedra.*

for $1 \leqq i \leqq n$. The shortest path between successive points $(i, i, 0)$ does not pass through the apex of the pyramid because the path along the x, y-plane is shorter. Hence, by symmetry, there are two distinct shortest paths from $(i, i, 0)$ to $(i+1, i+1, 0)$ for each i. This implies that there are 2^{n-1} shortest paths from $(0, 0, 0)$ to $(n, n, 0)$. Each shortest path has a different edge sequence.

The exponential number of paths is due to the fact that, at nonconvex vertices, shortest paths may cross one another without restriction. Given the similarity of the shortest path structure on nonconvex polyhedra with convex polyhedra [3], it is natural to conjecture that if the set of shortest paths on nonconvex polyhedra are suitably restricted (for example, allowing exactly one shortest path between any pair of vertices), then the proof given here can be generalized to nonconvex polyhedra. However, one major complication is the fact that arbitrary nonconvex polyhedral surfaces may not be planar.

6. Concluding remarks. We have shown that the number of shortest path sequences on the surface of a convex polyhedron is at most $O(n^4)$ and that this bound is asymptotically tight. We proved this result in the more general setting of pseudosegments on a planar subdivision, exploiting only topological properties of shortest paths. The principal question raised by this research is how to compute the actual edge sequences efficiently. Sharir showed how to compute the edge sequences in $O(n^8 \log n)$ time [5], and this has been improved to $O(n^7 \log n 2^{\alpha(n)})$ by Schevon and O'Rourke [4]. Does there exist an algorithm closer to $O(n^4)$, or better, an output sensitive algorithm whose running time is a function of the number of edge sequences? Another question is whether there is a suitably restricted definition of edge sequence from which we can show that the number of edge sequences on a nonconvex polyhedron is polynomial (perhaps a function of the genus of the polyhedron).

Acknowledgments. The author would like to thank one of the anonymous referees for many valuable comments, which greatly improved the clarity of the paper.

REFERENCES

[1] B. GRÜNBAUM, *Arrangements and spreads*, Confer. Board of the Math. Sciences, Reg. Confer. Series in Math., No. 10, American Mathematical Society, Providence, RI, 1972.

[2] 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.

[3] J. S. B. MITCHELL, D. M. MOUNT, AND C. H. PAPADIMITRIOU, *The discrete geodesic problem*, SIAM J. Comput., 16 (1987), pp. 647–668.

[4] C. SCHEVON AND J. O'ROURKE, *The number of maximal edge sequences on a convex polygon*, Tech. Report JHU-88/03, Johns Hopkins University, Baltimore, MD, June 1988.

[5] M. SHARIR, *On shortest paths amidst convex polyhedra*, SIAM J. Comput., 16 (1987), pp. 561–572.

[6] M. SHARIR AND A. SCHORR, *On shortest paths in polyhedral spaces*, SIAM J. Comput., 15 (1986), pp. 193–215.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 612-626, August 1990

002

IMPROVING THE TIME COMPLEXITY OF MESSAGE-OPTIMAL DISTRIBUTED ALGORITHMS FOR MINIMUM-WEIGHT SPANNING TREES*

F. CHIN† AND H. F. TING‡

Abstract. A distributed algorithm is presented that constructs the minimum-weight spanning tree of an undirected connected graph with distinct node identities. Initially, each node knows only the weight of each of its adjacent edges. When the algorithm terminates, each node knows which of its adjacent edges are edges of the tree. For a graph with n nodes and e edges, the total number of messages required by this algorithm is at most $5n \log n + 2e$, where each message contains at most one edge weight plus $3 + \log n$ bits. Although the algorithm presented here has the same message complexity as the previously known algorithm due to Gallager, Humblet, and Spira [*ACM Trans. Programming Language and Systems*, 5 (1983), pp. 66-77], the time complexity of the algorithm presented improves from Gallager's $O(n \log n)$ to $O(n \log^* n)$ time units, where $\log^* k$ is the number of times the log function must be applied to k to obtain a result less than or equal to one. A worst case of $\Omega(n \log^* n)$ is also possible. In addition, when the network is synchronous, the algorithm presented is modified further to solve the same problem with the same message complexity but in $O(n)$ time.

Key words. distributed algorithms, synchronous and asynchronous networks, minimum spanning trees, communication complexity

AMS(MOS) subject classifications. 68M10, 68Q25

1. Introduction.

Given an undirected connected graph G with n nodes and e edges, where each node has a unique identity, a **spanning tree** of G is a connected subgraph of G with exactly n nodes and $n-1$ edges. The **weight of a spanning tree** is the sum of weights of all edges in the spanning tree. Our problem is to design a distributed algorithm that finds a spanning tree of G whose weight is minimum, i.e., the **minimum-weight spanning tree** (MST) of G.

We assume that a processor exists at each node of the graph, and the processor initially knows the weights of the edges adjacent to the node. Each node performs the same local algorithm and two adjacent nodes can communicate with each other by exchanging messages on the edge between them. A node can send (broadcast) or receive messages on several adjacent edges simultaneously. Messages can be transmitted independently in both directions on an edge, without error and in sequence. In an asynchronous network, each message sent by a node to any of its neighbors arrives within some finite but unpredictable time. However, in the synchronous network, there is a global clock accessible by all nodes, and messages are allowed to be sent only at integer pulses of the clock. During each clock pulse at most one message can be sent over a given edge and the delay of each message is at most one time unit (i.e., one pulse duration) of the global clock.

The **time complexity of a synchronous algorithm** is defined as the maximum number of clock pulses passed between the sending of the first and the receiving of the last message of the algorithm. In an asynchronous network, we assume the existence of a hypothetical global clock. The processing and queueing time of each message is negligible, and the transmission of each message takes at most one time unit. The time

* Received by the editors January 20, 1987; accepted for publication (in revised form) September 15, 1989.
† Department of Computer Science, University of Hong Kong, Pokfulam, Hong Kong.
‡ Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

complexity of an asynchronous algorithm is the maximum number of time units from the start to the completion of the algorithm. This assumption is introduced only for the purpose of performance evaluation and the algorithm can operate correctly with arbitrary delays.

Since Minimum-Weight Spanning Tree (MST) is one of the most fundamental structures of a graph, it is not surprising that the MST construction algorithm can serve as building block for many other distributed algorithms, such as network synchronization [2], breadth-first-search [3] and deadlock resolution [4]. Awerbuch has proved in [5] that this problem is equivalent to a large class of problems (e.g., leader selection, spanning tree construction, counting the number of nodes in a network and computing a sensitive decomposable function).

To the best of our knowledge, all distributed algorithms that solve the MST problem [5], [11], [12], [15], [18] employ the idea of Borůvka [7], [8], [14], the so-called Sollin algorithm. A **fragment** is defined to be a subtree of a MST. Initially each node is treated as a fragment, and fragments are merged together iteratively over their minimum-weight outgoing edges. An edge e is an **outgoing edge** of a fragment if one of its endpoints is in the fragment and the other is not. The algorithm terminates when one fragment remains. Each fragment finds its minimum-weight outgoing edge independently and has a designated edge called the **core** to coordinate action. The algorithm proceeds in phases in which fragments are merged into larger ones. During each phase, information must be broadcast from the core to every node in the fragment and vice versa. Unfortunately, the message complexity of the obvious implementation of this algorithm is $\theta(n^2)$. Message complexity is worst when there is a large fragment (say a fragment with $n/2$ nodes) that goes through $n/2$ phases by enlarging its size one node at a time. Since during each phase at least $n/2$ messages are needed for the communication between the core and all other nodes, $\theta(n^2)$ messages are needed for the algorithm.

Gallager, Humblet, and Spira [13] later proposed an improved algorithm that solves this problem in $\theta(e+n\log n)$ messages. When two fragments are merged, Gallager's algorithm ensures that the small fragment is merged into the large one and work is only done by the small fragment. Since each fragment is always merged into a fragment of at least its original size, each node can go through at most $\log n$ merges. To implement this mechanism, a **level** field is associated with each fragment, with the property that a fragment at level l has at least 2^l nodes; in particular, fragments with a single node have $l=0$. The level of a node is defined as the level of its containing fragment. Thus, whenever two fragments at different levels join, the fragment at the smaller level works harder, never waits, and assumes the name and level of the larger. On the other hand, if a higher level fragment tries to merge with a lower level one, the former fragment waits until the latter fragment reaches a level high enough for combination. When two fragments at the same level join, both fragments go through the name change and a new fragment with its level increased by one is formed.

As given in [13], $\Omega(e)$ messages are necessary in constructing a spanning tree, and as proved in [16], $\theta(n\log n)$ messages are needed to find a leader on a ring. It follows that $\Omega(e+n\log n)$ messages are necessary to construct a spanning tree in a general network. Thus, Gallager's algorithm is message-optimal. However its worst-case time complexity is $\theta(n\log n)$. Let us consider a fragment of size $n/2$ but at a low level, say $l=1$. Because of its initially low level, this fragment can go through $\log n$ merges before forming the final MST. Since each merge may take $\theta(n)$ time to update the information of the fragment, Gallager's algorithm might require $\theta(n\log n)$ time. This happens because the size of a fragment is not reflected by its level. To be more

specific, when one fragment joins another, high-level fragments must wait for low-level ones to respond. But the size of those low-level fragments may be large and messages may have to travel from one end of the fragment to the other end, in order to update the level and identity of the fragment and to find its minimum-weight outgoing edge. Thus, the waiting time can be unduly long.

In this paper, we modify Gallager's algorithm by introducing the property that a fragment at level l has size bounded within 2^l and 2^{l+1}, otherwise its level will be updated. Thus, if there exists a large fragment, its level will be increased accordingly and its workload can therefore be reduced. The message complexity of this algorithm remains $5n \log n + 2e$ but the time complexity can be reduced to $15n \log^* n + 3n$. The detailed description of our algorithm is given in § 2. In fact, similar observations and results have also been obtained independently by Gafni [11]. In § 2.4, an example that uses $\Omega(n \log^* n)$ time is presented.

In § 3, we further modify the algorithm for a synchronous network so that a high-level fragment can immediately terminate its waiting for another low-level fragment's response if the waiting time of the high-level fragment is unduly long. Should this happen, the size of these low-level fragments must be large enough for merging and thus the waiting of the high-level fragment can be terminated and be merged with the low-level fragment correctly. Since the durations of all waitings that depend on the levels of the corresponding fragments are bounded, the algorithm can be executed in synchronized phases and terminates in $\theta(n)$ time. Recently, Awerbuch [5] has shown that the asynchronous MST problem can also be solved in $\theta(n)$ time with $O(e + n \log n)$ messages.

2. The asynchronous algorithm. Since our algorithm depends very much on the one given by Gallager, Humblet, and Spira [13], we adopt similar notation in our algorithm description. In particular, uppercase words stand for labels and states, whereas italic words are for messages. Our algorithm starts with all n nodes awake. This assumption can be relaxed by propagating an *AWAKE* message to all nodes initially. This can be done with at most $n-1$ extra time units and $2e$ extra messages. Initially, each node is a fragment at level zero. Fragments are merged together iteratively as described in Gallager's algorithm. However, the method for determining a fragment level and the mechanism for merging fragments are somewhat different.

As given in [13], we have the following definitions and properties:

(1) A node has two possible states: **FIND** and **FOUND**. Initially, all nodes are in the FOUND state.

(2) An edge has three possible states. Initially, all edges are in the **BASIC** state. An edge is in the **SELECTED** state if it is found to be a MST edge. It is in the **REJECTED** state if it is known not to be a MST edge.

(3) Every fragment usually has a SELECTED edge as its core. The adjacent nodes of the core act as the coordinators. For those fragments that do not have a core, a node is designated as the coordinator.

(4) Every fragment has a fragment level and a **fragment identity**, which equals to the weight of either its core or an edge incident to its coordinator (if there is no core in the fragment).

2.1. How a fragment finds its minimum-weight outgoing edge. Whenever a new fragment is formed, the coordinator(s) of the fragment broadcasts the message *INITIATE*(*F*, *l*, FIND) along the SELECTED edges of the fragment. As this set of SELECTED edges forms a spanning tree of the fragment, all the nodes in the fragment are

informed about their new fragment identity F and their new fragment level l. At the same time, all the nodes change their states to FIND in order to participate in finding the fragment's minimum-weight outgoing edge.

When a node u enters the FIND state, it finds the minimum-weight outgoing edge by sending a *TEST*(F, l), message over its minimum-weight BASIC edge to, say, node u' in fragment F' at level l', and waits for a response. If the response message is *REJECT*, i.e., F and F' have the same identity, then u marks the edge REJECTED and sends a *TEST* message over its next minimum-weight BASIC edge. The *TEST* message will be sent until either an *ACCEPT* message is received or there are no more BASIC edges adjacent to u. If u receives an *ACCEPT* message on a BASIC edge, there it will remember the edge as **min_edge** and its weight as **min_weight**. On the other hand, if there is no outgoing edge from u, min_weight is set to infinity.

When a node u' in fragment F' at level l' receives a *TEST*(F, l) message on a BASIC edge, node u' responds with a *REJECT* message and marks the edge as REJECTED if F and F' are equal. If F and F' are different and $l \leqq l'$, then u' responds to u immediately with an *ACCEPT* message. On the other hand, if $l > l'$, u' will wait until $l' \geqq l$.

A node u in state FIND will eventually send a *REPORT*(W, SZ) message along the SELECTED edges to its coordinator, where W stands for the weight of the minimum-weight outgoing edge and SZ for the size of the subfragment root at u. Assume W_i and SZ_i are the minimum-weight outgoing edge and the size of the subfragment rooted at u's ith son. If u is a leaf node, $W = u$'s min_weight and $SZ = 1$, otherwise, $W =$ the minimum of min (W_i) and u's min_weight and $SZ = 1 + \sum SZ_i$. At the same time, node u **marks** the minimum-weight outgoing edge of the subfragment rooted at itself, i.e., its min_edge or the edge leading to its son that has the minimum W_i. Node u sends a *REPORT*(W, SZ) message to its father and changes its state to FOUND only after it has found its min_weight and has received all *REPORT*(W_i, SZ_i) messages from its sons (if any). When the coordinator(s) of a fragment receives all *REPORT* messages from its sons, the fragment size and the weight of the minimum-weight outgoing edge can be determined.

The algorithm terminates when the returned value W in the *REPORT* message at the coordinator is infinity. This implies there are no outgoing edges and there is only one fragment in the graph.

As shown in [13], when the coordinators receive all the *REPORT* messages, it must be the case that $SZ \geqq 2^l$ where l is the fragment level. But since the time the fragment initiated the process of finding its minimum-weight outgoing edge, other fragments may have merged into it and its fragment size and level may not be commensurate, i.e., $SZ \geqq 2^{l+1}$. As we must make sure that the fragment level always reflects its size, the coordinator(s) compares SZ with 2^{l+1}. If $SZ \geqq 2^{l+1}$, the coordinator(s) broadcasts another *INITIATE*$(F, l'$, FIND$)$ message to all nodes in the fragment to update their level as if a new fragment at level l' has just formed, where l' is the largest integer such that $SZ \geqq 2^{l'}$. Note that this level-updating process may be repeated many times until the fragment size and level are commensurate (i.e., $2^l \leqq SZ < 2^{l+1}$). On the other hand, if $SZ < 2^{l+1}$, then the fragment is ready to combine with another fragment to form a new fragment with a new core. The coordinator(s) sends a message *CHANGECORE* following the path of the *marked* edges to the node adjacent to the minimum-weight outgoing edge.

2.2. How fragments are merged together. When the *CHANGECORE* message in a fragment at level l reaches node u, to which the minimum-weight outgoing edge

(u, u') is incident, u attempts to merge its fragment with the fragment F' at level l' that contains u' by sending a *CONNECT*(l) message over (u, u'). After receiving this *CONNECT*(l) message, u' compares l with its fragment level l'. There are two possible outcomes, $l = l'$ or $l < l'$. The outcome $l > l'$ is not possible because levels are nondecreasing and a *TEST* message must have been sent and responded to before this *CONNECT*(l) message is sent.

Case 1. $l = l'$. If u' has previously sent a *CONNECT*(l') message over edge (u, u') to u, then the two fragments will have the same minimum-weight outgoing edge, and will be merged immediately together to form a new fragment F'' at level $l+1$ with edge (u, u') as its new core. Edge (u, u') is marked SELECTED. *INITIATE*$(F'', l+1,$ FIND) messages are then broadcast to all nodes in F''. In all other cases, u' will wait until it sends a *CONNECT*(l') message to u and proceeds as previously described, or until it increases its level l', in which case it does the following.

Case 2. $l < l'$. Normally, fragment F is merged with fragment F'. Because of our strategy of never making a low-level fragment wait, node u' immediately marks edge (u, u') as SELECTED and sends an *INITIATE*(F', l', S') message to u, where F', l', and S' stand for fragment identity, level, and state, respectively, of u'.

If $S' =$ FIND, then u' has not sent its *REPORT* message. Fragment F simply joins fragment F' and participates in finding the minimum-weight outgoing edge of the enlarged fragment. Node u marks edge (u, u') as SELECTED, changes its state to FIND, fragment identity to F', fragment level to l', and also relays the *INITIATE*(F', l', S') message to all other nodes in F. Meanwhile, u' waits for the *REPORT* message from u before sending its *REPORT* message.

If $S' =$ FOUND, then u' has already sent its *REPORT* message. Thus the size of fragment F cannot be reported to the coordinator of F'. However, we want to make sure that its size is reflected at the next level update. Under such circumstances, fragment F will not be merged with F' immediately but instead will be treated as a new fragment with a new identity and a new level. Basically, node u will be the new coordinator of the fragment. It changes its fragment identity to ω, the weight of a SELECTED edge incident to u, its state to FOUND, its level to l', broadcasts an *INITIATE*$(F, l',$ FOUND) message with $F = \omega$ to all ther nodes in the fragment, and waits for their *REPORT* messages. As edge (u, u') remains the minimum-weight outgoing edge for this new fragment, state FOUND is assigned to all nodes in the fragment. Note that u will not mark edge (u, u') as SELECTED, whereas on the other hand, u' has already sent its *REPORT* message and has marked edge (u, u') as SELECTED. Thus, messages can still be transmitted from u' to u as if they are in the same fragment, so it is possible for u to receive another *INITIATE* message from u' before u receives all its sons' *REPORT* messages. In order to ensure that every *INITIATE* message has been reported before another *INITIATE* message is issued, node u may have to delay its action on the second *INITIATE* message until it has received all the *REPORT* messages from its first *INITIATE* message. Hence, there is at most one such pending *INITIATE* message from u'.

Having received all its sons' *REPORT* messages, u compares its fragment size SZ with $2^{l'+1}$. F and F' are combined together only when $SZ < 2^{l'+1}$. In other words, when the size of F is small enough and is reflected by its new level, the size of F does not have to be reported to the coordinator of F' and F can be merged into F' without problem. On the other hand, if the size of F is sufficiently large, F will not be combined with F'. In order to prevent a large fragment from merging with a small one, we delay the process in F and make F wait by increasing its level sufficiently. Thus there are two cases:

(a) $SZ < 2^{l'+1}$. Fragment F can be absorbed into fragment F'. Node u will mark edge (u, u') as SELECTED. If there is a waiting *INITIATE* message, u will process the second *INITIATE* message as if it had just been received. Even though F and F' have different identities in this merged fragment for an uncertain period before this second *INITIATE* message is processed in F, this does not create any problems in checking whether an edge is an outgoing edge from F' to F. This is because if a *TEST* message is sent over an edge, say (v', v) from F' to F, then the fragment level of v will be less than that in the *TEST* message (i.e., the fragment level of v'). Node v would delay making any response until it receives the second *INITIATE* message and obtains the same fragment identity and level as node v'. Thus, that *TEST* message will be rejected eventually.

(b) $SZ \geqq 2^{l'+1}$. Let l'' be the largest integer such that $SZ \geqq 2^{l''}$. Node u will change its level to l'', its state to FIND, and will send a *REPORT*$(w, 0)$ to u', where w is the weight of the edge (u, u'). Furthermore, node u will broadcast the *INITIATE*(F, l'', FIND) to all its sons as if a new fragment has just been formed.[1] On receiving a *REPORT*(w, x) message with $x = 0$ over a SELECTED edge, node u' remarks that edge as BASIC and handles the *REPORT* message as usual. From then on, fragment F and F' are treated as separate fragments.

2.3. Analysis of the algorithm. The correctness proof and message complexity analysis are same as given in Gallager, Humblet, and Spira [13]. For the correctness proof, we only need to prove that in our modified algorithm, deadlock will not be created and an edge is a branch of the MST if and only if it is SELECTED. From the description of our algorithm in the previous section, our only modifications are to raise the level of a fragment when its size is too large and to delay the merging of the fragments when its resultant size cannot be reflected by its level. However, these modifications would not change the fact that only the minimum-weight outgoing edge of a fragment will be marked SELECTED and that the smallest-level fragments never wait. With the same argument as in [13], fragments remains to be subtrees of the MST and there is no deadlock in the algorithm.

As far as the message complexity is concerned, it seems that more messages are required to ensure the size of each fragment is reflected by its level. However, this increase of messages is always associated with a level change of a fragment and each node at any given level still transmits/receives at most five messages [13]. Since fragments are always merged into larger fragments and their levels only increase, a node can go through at most $\log n$ levels and this accounts for the $O(n \log n)$ messages. With the fact that an edge can be rejected only once, and each rejection requires two messages, there are at most $2e$ messages leading to rejections. Thus, the total message complexity remains $O(e + n \log n)$.

In order to prove the upper bound on time complexity, we have the following definitions. A **fragment** F is the largest minimum-spanning subtree whose vertices have the fragment identity F. A **subfragment** of F is a fragment whose next fragment identity is F. If $F_1, F_2, \cdots, F_m$ are all the subfragments of F, it can be shown easily that all F_i's are disjoint, $F = \bigcup_{i=1}^{m} F_i$, and $SZ = \sum_{i=1}^{m} SZ_i$, where SZ and SZ_i stand for the sizes of the fragment F and F_i, respectively.

For $0 \leqq i \leqq \log n$, let τ_i be the minimum time of the hypothetical global clock at which all nodes in the graph have level at least i and $a_i = \tau_i - \tau_{i-1}$. A **critical node** at

[1] We now briefly describe a modification of the algorithm that is used to improve its complexity. If the waiting *INITIATE* message is of level at least l'', then it can be processed immediately without broadcasting the *INITIATE*(F, l'', FIND) message to all the nodes in F.

τ_i is the node that changes from a level less than i to a level at least i at time τ_i. A **critical fragment** at τ_i is the fragment containing a critical node at τ_i at time τ_i. Note that there may be several critical nodes at τ_i, and likewise, several critical fragments. Since all the critical fragments are disjoint and the same argument can be applied to them, without loss of generality, let us consider a particular critical node and the corresponding critical fragment. An **active node set**, AN_i, at τ_i is the set of nodes in the critical fragment at τ_i with level less than i at time τ_{i-1}. From the definition of τ_{i-1}, these nodes are at level $i-1$. If the active node set at τ_i is nonempty, this set of nodes may belong to several subfragments at level $i-1$ of the critical fragment at τ_i. As all nodes are initially AWAKE, $\tau_0 = 0$. If l is the fragment level when the algorithm terminates, then $\tau_i = \tau_l$ for $l \leqq i \leqq \log n$. Define,

$$\log^{(k)} n = \begin{cases} \log n & \text{if } k = 1, \\ \log (\log^{(k-1)} n) & \text{if } k > 1. \end{cases}$$

It is obvious that $\tau_{\log n} = \sum_{i=1}^{\log n} a_i$. We now partition the indices i into sets. Let $S(k)$ be the index set $\{i: (\log n - \log^{(k)} n) < i \leqq (\log n - \log^{(k+1)} n)\}$. For example, $S(1) = \{1, 2, \cdots, \log n - \log^{(2)} n\}, \cdots, S(\log^* n - 2) = \{\log n - 3, \log n - 2\}$, $S(\log^* n - 1) = \{\log n - 1\}$ and $S(\log^* n) = \{\log n\}$. The a_i's are partitioned into classes according to these index sets, such that all the a_i's whose indices belong to the same index set are in the same class. We want to show that the sum of all a_i's in any class is $O(n)$. Since there are $\log^* n$ classes in all, we have $\tau_{\log n} = O(n \log^* n)$.

Let us consider the kth class. The elements i in $S(k)$ are partitioned into two groups, $S'(k)$ and $S''(k)$, according to the value of the corresponding a_i's. $S'(k)$ contains the indices of all the small a_i's and $S''(k)$ contains the indices of the large ones; more precisely, $S'(k) = \{i \in S(k): a_i \leqq 10n/\log^{(k)} n\}$ and $S''(k) = S(k) - S'(k)$. Since $S(k)$ contains $(\log^{(k)} n - \log^{(k+1)} n)$ elements and $S'(k) \subseteq S(k)$, the sum of all the small a_i's whose indices are in $S'(k)$ is no more than $O(n)$. As for the large a_i's corresponding to $S''(k)$, we will prove that the sizes of their corresponding AN_i's, and their corresponding critical fragments, cannot be small. Intuitively, this can be seen as follows. If the AN_i corresponding to a large a_i is small, then the time required to change the level of these nodes in AN_i's is small, contradicting the fact that a_i is large. Since our algorithm makes sure that sizes of fragments are reflected by their levels, the level of each fragment containing a large AN_i will be increased accordingly. As AN_i is large, so is its level increase. Thus, the level of the nodes in these large AN_i's becomes so high that these nodes cannot be in another AN_j in the same class as a_i. Thus, there cannot be too many large a_i's in the kth class corresponding to $S''(k)$, and consequently the sum of the large a_i's whose indices are in $S''(k)$ is also $O(n)$. We now make these observations precise.

LEMMA 1. *$a_i \leqq 5|AN_i|$, where a_i is the time required to increase the level of all nodes with level less than i at τ_{i-1}, which includes the active node set AN_i, to a level at least i.*

Proof. The lemma holds true for $a_i = 0$. Assuming $a_i > 0$, we have $\tau_i > \tau_{i-1}$ and $AN_i \neq \varnothing$. At time τ_{i-1}, every node v in AN_i has received the *INITIATE* message at level $i-1$ and is ready to send or has sent some *TEST* messages. Let us consider the situation when v must send some *TEST* messages to find its min_weight. All nodes in the fragment to which v belongs at time τ_{i-1} must be at level $i-1$, and the size of this fragment is at most $|AN_i|$. Hence, v will send at most $|AN_i|$ *TEST* messages and will receive at most $|AN_i| - 1$ *REJECT* messages before an *ACCEPT* message is received. As v's level is the lowest in the graph, the *TEST* messages will be responded to without any delay, and thus after at most $2|AN_i|$ time units, all the leaf nodes in AN_i relative to the tree corresponding to the critical fragment can report. As there are

$|AN_i|$ active nodes, there will be at most $|AN_i|$ *REPORT* messages, $|AN_i|$ *CHANGECORE/CONNECT* messages, and $|AN_i|$ *INITIATE* messages from within the critical fragment. Thus, it takes at most $3|AN_i|$ extra time units before all nodes in AN_i rise to a level at least i. Thus, we have $a_i \leqq 5|AN_i|$. □

COROLLARY 1. *If $a_i > 5m$, then $|AN_i| > m$, where m is any positive integer.*

LEMMA 2. *Let v be a node in fragment F at level l at time t. On its next level update, v will change its level from l to l' such that $2^{l'+1} > |F|$.*

Proof. The proof follows directly from the description of the algorithm. The size of the fragment must be reported after every *INITIATE* message, which is the only way to change the level of a fragment. The new level is assigned according to the fragment size. □

Let $R(k) = \sum_{i \in S(k)} a_i$ for $1 \leqq k \leqq \log^* n$. Then $\tau_{\log n} = \sum_{k=1}^{\log^* n} R(k)$.

LEMMA 3. *$R(k) \geqq 15n$ for $1 \leqq k \leqq \log^* n$.*

Proof. $S(k)$ is partitioned into $S'(k)$ and $S''(k)$. $R(k) = R'(k) + R''(k)$ where

$$R'(k) = \sum_{i \in S'(k)} a_i \quad \text{and} \quad R''(k) = \sum_{i \in S''(k)} a_i.$$

Since $a_i \leqq 10n/\log^{(k)} n$ for all $i \in S'(k)$, we have

$$R'(k) = \sum_{i \in S'(k)} a_i \leqq \left(\frac{10n}{\log^{(k)} n}\right)(\log^{(k)} n - \log^{(k+1)} n) \leqq 10n.$$

As $a_i > 10n/\log^{(k)} n$ for all $i \in S''(k)$, from Corollary 1 we have $|AN_i| > 2n/\log^{(k)} n$. The initial level of a new fragment may not reflect its size, but by Lemma 2, all nodes in AN_i will raise their level to greater than $\log n - \log^{(k+1)} n$ at their next level update. From then on, by the definition of $S(k)$, these nodes will never belong to any any other active node set corresponding to $S(k)$, i.e., they can only belong to some active node set AN_j in some $S(k')$, with $k' > k$. Thus, before this level update, all these nodes in the graph may belong to at most one active node set corresponding to $S''(k)$. Using this fact and Lemma 1, we have $R''(k) = \sum_{i \in S''(k)} a_i \leqq \sum_{i \in S''(k)} 5|AN_i| \leqq 5n$. Hence $R(k) = R'(k) + R''(k) \leqq 15n$. □

Using Lemma 3 and the equality $\tau_{\log n} = \sum_{k=1}^{\log^* n} R(k)$, we have the following theroem.

THEOREM 1. $\tau_{\log n} \leqq 15n \log^* n$. □

After $\tau_{\log n}$ and before the algorithm terminates, at most $3n$ time units are required for the *TEST*, *REJECT*, and *REPORT* messages as desdcribed in the proof of Lemma 1. Thus, our algorithm takes no more than $15n \log^* n + 3n$ time units.

2.4. Example with time complexity $\Omega(n \log^* n)$. Let us consider two extreme situations. If there are many small fragments merging in pairs, each node may participate in $\log n$ level changes before the algorithm terminates. However, under this situation, our algorithm allows a large amount of parallelism in message exchanges and each fragment doubles its size at each merging or level change. Since the time delay for each level change is proportional to the fragment size, the total time delay of the algorithm would still be $O(n)$. If, on the other hand, there is a large fragment, our algorithm guarantees that the fragment's level will be raised appropriately. Even though there might be a long delay for this level update, this time loss can be compensated by the large level increase and the algorithm can still be finished in $O(n)$ time. We can show that the worst time complexity of our algorithm occurs when there is a sequential waiting of the fragments with size $n/\log n$, and after $O(n)$ time, the number of fragments can at most be reduced from n to $\log n$. This process can be repeated for $\log^* n$ times until a single fragment remains. Here is a sketch of our example whose

execution requires at least $\log^* n$ stages with each stage takes $\Omega(n)$ time. Thus, the whole execution of our example requires $\Omega(n \log^* n)$ time.

Let us first consider $\log k$ rows of nodes, each row has k nodes, connected in a line as shown in Fig. 1, which is an example with $k=8$. The nodes in each row are further subdivided into groups; the ith row is divided into $k/2^{i-1}$ groups, each with 2^{i-1} nodes (the nodes in each oval form a group). The nodes in each group are linked together with edges of lowest weight, and thus they will first merge together to form supernodes of level $i-1$ (for nodes in the ith row). Furthermore, these groups are connected in a line with edge weights strictly ascending from left to right. When the minimum-weight spanning tree algorithm is executed, the nodes in the ovals form supernodes in each row. Let us consider the supernodes of the ith row where $i>1$. They are at level $i-1$ and their leftmost supernode sends a *TEST* message over an edge of the highest weight (the 1,000 link in Fig. 1) to a supernode of the $(i-1)$st row at level $i-2$. This *TEST* message will be responded to when all the nodes in the $(i-1)$st row have merged into a single fragment with its level increased to $\log k$. Then, the leftmost two supernodes in the ith row will merge together to form the core of a new fragment. The other supernodes in the same row will then merge one after another without concurrency in a strictly left to right fashion. Finally, all the nodes in each row will merge into a fragment of level $\log k$, with the leftmost intersupernode edge as the core.

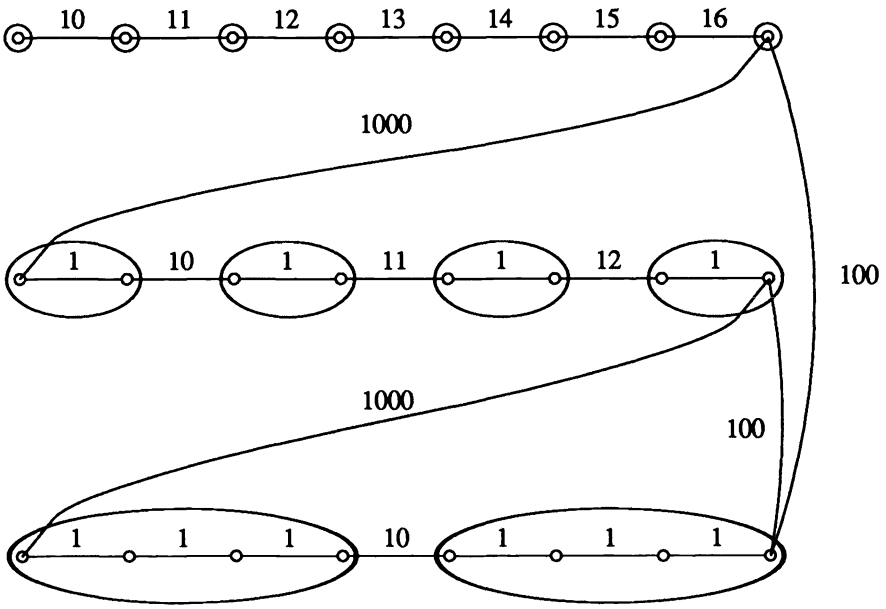


FIG. 1

Although each row will eventually form a single fragment, this cannot happen until all the nodes in the preceding row have merged together and have had their levels updated to $\log k$. Since there are $\log k$ rows and the level update of all the nodes in each row takes $\theta(k)$ time units, it takes $\theta(k \log k)$ time for all the rows to become fragments. If we let k be about $n/\log n$, then it takes $\theta(n)$ time for all the rows to form single fragments.

Let F_i be the fragment formed by the ith row. F_i is at level $\log k$ and, except for the last row, sends a *TEST* message over the edge of the second highest weight (the 100 links in Fig. 1) to a node at the last row which is at level $(\log k - 1)$. Since that *TEST* message goes to a fragment at a lower level than F_i, F_i must wait until the last row has formed a single fragment before connecting to another fragment. Basically, these second highest weight edges (the 100 links) are for the purpose of synchronizing each stage.

The above construction is repeated for several stages by treating each F_i as a supernode. At each stage, the number of supernodes is the logorithm of the number of nodes (supernodes) in the preceding stage. So, this whole process will be done in $\Omega(\log^* n)$ stages and it will take $\Omega(n \log^* n)$ time to complete the entire algorithm. In order that the execution of each stage is not affected by what is done in the preceding stages, without loss of generality, let us consider the second stage. Two new nodes a, b are introduced for each fragment F_i, and they are connected with edges of weight x, which is larger than those on any of the edges of but less than those edges between the F_i's, as in the Fig. 2.

F_i

FIG. 2

Let the two new nodes a, b be the **left** and **right** *handles* of F_i, respectively. By making the edges connected to the handles are of weight larger than x, the handle will join the subfragments of F_i at an early stage, and will have no effect on the preceding stage. In addition, any edges attached to the handles will not affect the execution of the preceding stage either. In other words, F_i is formed in the first stage even with the handles. Fragments F_i's are then connected into the same structure as in Fig. 1 by treating each F_i, together with its handles, as a supernode.

With k, the number of nodes (fragments) in each row, equals to $\log n / \log \log n$, there will be enough fragments F_i. The supernodes are then connected in such a way that edges coming into the left of the supernode are attached to its left handle, and those connected to the right are connected to the right handle to make sure that the second stage could not start until the first stage is complete. The weights of the new edges will all be larger than x and in the mid-range (say 50, if Fig. 1 is used as an indicator of weights in the preceding stage). That is, they will be larger than the weights of the branches within each F_i, and they will be less than the previous stage's edges between the F_i.

We start in the first stage the construction with k nodes in each row and $\log k$ rows. With $k = n/\log n$, there are about $(n/\log n)(\log n - \log \log n)$ nodes. Since each fragment F_i requires a pair of handles, it requires at most $2 \log n$ handles at the first stage. Similarly, $2 \log \log n$ for the second stage. Thus, there are a total of no more than $2(\log n + \log \log n + \cdots)$ nodes added. So for large n, the number of nodes in the final graph is less than n. Note that the graph can be padded out to just n nodes in any reasonable fashion.

3. The synchronous algorithm. Let us consider the above algorithm in more detail and understand why the nonlinear time bound still exists. The worst-case time complexity occurs when there is sequential waiting of medium-size fragments. The nonlinear characteristic of the time complexity is due to the fact that the level of a fragment

cannot be increased gradually step by step. If many mergings to fragment F occur almost at the same time, there may be a large level increase of fragment F, say from l to l', after $2^{l'}$ time units. In the meantime before F changes its level to l', another fragment F'' at a slightly higher level than F, say l'', where $l < l'' \ll l'$, may have sent a *TEST* message to fragment F and is waiting for the level change of fragment F to l' in order to raise its own level from l'' to $l''+1$. Since the waiting can be unnecessarily long (up to $2^{l'}$ time units) for one unit of level increase from l'' to $l''+1$, the time complexity of the algorithm is nonlinear.

In this section, we propose a synchronous version of our previous algorithm for solving this MST problem in $\theta(n)$ time with the same $\theta(e+n \log n)$ message complexity. This algorithm is a modification of the algorithm by Gallager, Humblet, and Spira [13] and tackles the above problem with an approach different from that of Awerbuch [5]. Awerbuch's approach is to reimburse fragment F'' for its time loss by hooking fragment F'' onto fragment F and subsequently inheriting level l'. In this case, fragment F'' has got its reward through its long waiting. Instead of hooking fragment F'' with fragment F through the minimum-weight outgoing edge as in the traditional methods, an arbitrary edge is chosen and a spanning tree is found instead of a minimum-weight one.

In order to see how our synchronous MST algorithm can be finished in $O(n)$ time, without loss of generality, let us assume that all the nodes awake simultaneously and execute the algorithm at the same time. Let T_l be the minimum time[2] when all the fragments in the graph have sizes at least 2^l (clearly, $T_0 = 0$). Thus, if a *TEST* message is sent after T_l from fragment F of level l through an interfragment edge to another fragment F', the *TEST* message can be accepted immediately regardless of the current level of F'. On the other hand, if the *TEST* message is sent through an intrafragment edge and if fragment F is of size less than 2^{l+1}, all its node's identities and levels would have been updated by time $t_B + 2^{l+1}$, where t_B is the broadcast time of the *INITIATE* message by the coordinators and the *TEST* message would be rejected (assuming that the *TEST* message is sent after $t_B + 2^{l+1}$). If, however, the size of fragment $F \geqq 2^{l+1}$, i.e., its size is not reflected by its level, the core will broadcast another *INITIATE* message and will command all the nodes to re-find the minimum-weight outgoing edge again. Based on this observation, if we modify our algorithm so that the *INITIATE* message is broadcast by the core only after T_l, i.e., $t_B > T_l$, and all the nodes participate in finding the minimum-weight outgoing edge after time $t_B + 2^{l+1}$, the *TEST* messages can be responded to immediately without considering the levels of the fragments, and the minimum-weight outgoing edge can be found correctly.

For every increase in fragment size, messages such as *INITIATE*, *REPORT*, and *CHANGECORE* may have to traverse from one end of the fragment to another, but this takes at most $O(2^{l+1})$ time units. Since all messages are responded to without delay, the time T_{l+1}, which guarantees that all fragments are of size at least 2^{l+1}, can be bounded by the inequality $T_{l+1} \leqq T_l + O(2^{l+1})$. Based on the recurrence formula for T_{l+1}, we can show that $T_{\log n} = O(\sum_{l=1}^{\log n} 2^l) = O(n)$, i.e., the time complexity of the algorithm is $O(n)$. We shall prove later that this recurrence inequality for T_{l+1} can also account for the fact that the nodes may not execute the algorithm simultaneously.

The algorithm can be described as follows. It starts by broadcasting *AWAKE* messages to all nodes across the whole network and any newly formed fragment starts

[2] Note that T_l is different from τ_l. In all cases $T_l \leqq \tau_l$ because a fragment is formed before all the nodes in the fragment obtain their fragment level and identity.

to find its minimum-weight outgoing edge by broadcasting an *INITIATE*(F, l, S, t_B) message from its core, where the arguments stand for the fragment identity, fragment level, fragment state,[3] and the broadcast time of the message. In order to guarantee the minimum size of fragments, we must have the broadcast time $t_B \geqq t_F + T_l + 2^{l+1}$ where t_F is the wake-up time of the fragment F, i.e., the minimum wake-up time of its coordinator(s) and T_l will be defined later. Each node v in the fragment then starts finding its own minimum-weight outgoing edge independently after time $t_B + 2^{l+1}$. This ensures that the process of finding the minimum-weight outgoing edge is synchronized among **all** nodes in the fragment.

Node v sends a *TEST*(F) message through its minimum-weight adjacent BASIC edge and waits for a response in exactly two time units. Note that there is no level argument in the *TEST* message and the response is immediate without considering their levels. The responded message is *REJECT* when F and F' have the same identity, and *ACCEPT* otherwise. Immediately after the minimum-weight outgoing edge of a node is found, *REPORT*(W, SZ) messages are reported to the core. *CHANGECORE* and *CONNECT* messages are sent as in Gallager's algorithm.

Note that if the level of the fragment cannot reflect its reported size, another phase of *INITIATE* messages may be needed. However, when fragment F' at level l' receives a *CONNECT* message from fragment F at level l, this situation can be slightly different because the case $l > l'$ is possible. Should this happen, fragment F waits until fragment F' has reached a level high enough for combination in order to guarantee that the message complexity be bounded by $O(e + n \log n)$, and then F' sends an *INITIATE* message to F as described in the previous algorithms.

As a digression, Awerbuch [6] defined an **H-partition** problem which is to partition a given graph into disjoint fragments such that each of which has at least H nodes. The construction of an H-partition for a graph is useful in a number of applications [1], [17]. If our algorithm terminates at T_l for $l \leqq \log n$, we can guarantee that all the fragments are of size at least 2^l. Thus, we can also solve the H-partition problem in $O(H)$ time and with $O(e + n \log H)$ messages as given in [6].

3.1. Correctness and complexity analysis. For proving the validity of our proposed synchronous algorithm, we must show in Lemma 4 that if T_l is defined as

$$T_{l+1} = \begin{cases} T_l + 28 \cdot 2^l, & \text{if } l \geqq 1, \\ 0 & \text{otherwise,} \end{cases}$$

then any fragment F with size SZ satisfying $2^{l-1} \leqq SZ < 2^l$ will correctly find its minimum-weight outgoing edge by time $t_F + T_l - 2^l$. Thus, we can guarantee that F would have merged into a larger fragment of size at least 2^l by time $t_F + T_l$.

LEMMA 4. *Any fragment F with size SZ satisfying $2^{l-1} \leqq SZ < 2^l$ can correctly find its minimum-weight outgoing edge for merging before time $t_F + T_l - 2^l$, where t_F is the wake up time of fragment F.*

Proof. The proof is by induction on the level l. Initially, every node has size $SZ = 1$, thus the hypothesis is true for $l = 1$. For the induction step, assume that the hypothesis is true for $l \leqq k$. Let us consider any fragment F, with size $2^k \leqq SZ < 2^{k+1}$. We want to show that the sizes of all the subfragments of F are less than 2^k, and thus we can apply the induction hypothesis to show that the subfragments can find their minimum-weight outgoing edge and merge together to form F within a bounded period,

[3] This state information FIND/FOUND can be omitted and this algorithm will also work with the assumption that each node is always in the FIND state.

i.e., $t_F + T_k + 2^{k+1}$. Since all the messages, such as *INITIATE*, *TEST*, *REJECT/ACCEPT*, *REPORT*, can be transmitted without delay and $SZ < 2^{k+1}$, we can show that the minimum-weight outgoing edge of F can be found within the time bound (i.e., $t_F + T_{k-1} - 2^{k+1}$ as stated in the hypothesis).

First, we shall prove by contradiction that all the subfragments of F are of size less than 2^k. Assume that there exists a subfragment F_i of F with size $2^k \leqq SZ_i < 2^{k+1}$. F_i will either merge into and obtain the level of another higher level fragment F_j, or merge with F_j over the same outgoing edge. For the former case, since $SZ_i \geqq 2^k$, $SZ_j \geqq 2^k$ and $F \supseteq F_i \cup F_j$, we have $SZ \geqq SZ_i + SZ_j \geqq 2^{k+1}$, which leads to a contradiction. For the latter case, it is required that we show that $t_c < t_{F_i} + T_k + 2^{k+1}$, where t_c is the time when a *CONNECT* message is sent from F_j to F_i. From our algorithm, the coordinator(s) of F_i will only broadcast the *INITIATE*(F_i, k, S, t_{B_i}) message at time t_{B_i}, where $t_{B_i} > t_{F_i} + T_k + 2^{k+1}$; thus we have $t_{B_i} > t_c$ if the inequality relation for t_c is true. This would imply that F_j has merged with F_i before F_i sends a *TEST*(F_i) message to F_j; this would then lead to a contradiction that F_i and F_j are merged together over the same outgoing edge. The remaining paragraph will prove the inequality relation for t_c. Since $SZ_j \leqq SZ - SZ_i < 2^k$, F_j will find its minimum-weight outgoing edge before time $t_{F_j} + T_k - 2^k$, from the induction hypothesis, and will take less than 2^k time units to send a *CONNECT* message to F_i; thus, $t_c < t_{F_j} + T_k$. As the distance between the coordinator of F_i and any nodes in F is less than 2^{k+1}, we have $t_{F_j} < t_{F_i} + 2^{k+1}$ and $t_c < t_{F_j} + T_k < t_{F_i} + T_k + 2^{k+1}$.

Using the argument in the previous paragraph, since all subfragments F_j of F are of sizes less than 2^k, t_c, the time when the *CONNECT* message is sent out from each F_j, will be strictly less than $t_{F_j} + T_k$. Since $t_{F_j} < t_F + 2^{k+1}$, we have $t_c < t_F + T_k + 2^{k+1}$, which in term is also less than F's broadcast time of the *INITIATE*(F, k, S, t_B) messages, t_B. Thus, all nodes in F would have received their corresponding fragment identity no later than $t_B + 2^{k+1}$. As from the algorithm, all the *TEST* messages are only sent after time $t_B + 2^{k+1}$, no intrafragment edge will be accepted and the minimum-weight outgoing edge can be correctly found.

Now, let us find an upper bound for the time when F finds its minimum-weight outgoing edge. Let k_0 be the level of F when it is initially formed. Without loss of generality, let us assume that before F determines its minimum-weight outgoing edge, there is a series of level updates from k_0 to k_1, to $k_2, \cdots$, and finally to k. Because there are at most 2^{k_i+1} *INITIATE*, *TEST*, *REJECT/ACCEPT*, *REPORT* messages for the ith level update and all the messages are transmitted without delay, each level update will not be broadcast after $t_F + T_k + 2^{k+1}$, the time when all subfragments would have been merged together, the total time for F to find its minimum-weight outgoing edge is less than

$$\begin{aligned} & t_F + T_k + 2^{k+1} + 4 \cdot (2^{k_1+1} + 2^{k_2+1} + \cdots + 2^{k+1} + 2^{k+1}) \\ & \quad < t_F + T_k + 2^{k+1} + 4 \cdot (2^1 + 2^2 + \cdots + 2^{k+1} + 2^{k+1}) < t_F + T_k + 26 \cdot 2^k \\ & \quad = t_F + T_{k+1} - 2^{k+1}. \qquad \square \end{aligned}$$

THEOREM 2. *The time complexity of our algorithm is* $55n$.

Proof. From Lemma 4, the algorithm terminates by time $t = t_F + T_{\log n+1} - 2^{\log n+1}$. Since $t_F < n$ and $T_{\log n+1} < 56n$, we have $t < 55n$. $\square$

The message complexity of our algorithm is $4n \log n + 4e$ rather than $5n \log n + 4e$ (including the initial $2e$ *AWAKE* messages). This is because *ACCEPT* messages are not needed in our algorithm; the reception of a *TEST* message can be assumed by default if no message is reported within two time units after it is sent. From Theorem

2, at most $\log n + 6$ bits are needed to encode the broadcast time. The fragment level can be encoded in $\log \log n$ bits and fragment size in $\log n$ bits. As three bits are enough to distinguish different types of messages, each message contains at most one edge weight or one node identity plus $(\log n + \log \log n + 9)$ bits.

4. Conclusion. We have improved the time complexity of the asynchronous and synchronous distributed algorithms for the minimum-weight spanning tree problem. Since our algorithms work for any arbitrary graph, it is obvious that $\theta(n)$ time complexity is needed for the problem. Our synchronous distributed algorithm for finding the minimum-weight spanning tree is not only message-optimal but also time-optimal. Even though Awerbuch has shown that $\theta(n)$ time is also achievable in the asynchronous network for this problem, it is not surprising to notice that a synchronous algorithm for a problem can always outperform its asynchronous counterpart [2], [9]-[11]. One of the open problems is to determine what extra information a synchronous algorithm has over an asynchronous one.

We have improved the time complexity of the Gallager, Humblet, and Spira's asynchronous algorithm for finding the MST of a graph to $O(n \log^* n)$. Although the worst-case message complexity remains $O(e + n \log n)$, our algorithms seem to have a larger message complexity because *REPORT* messages must be sent whenever a new fragment is formed. On the other hand, we can argue that our algorithms may use fewer messages on the average because whenever a large fragment is formed, its level will be increased sufficiently at the earliest possible instance and this may eliminate a number of unnecessary messages. It would be interesting to conduct some simulations to study the complexities of our algorithms when compared with Gallager's.

Acknowledgments. The authors thank M. Y. Chan for her careful reading and for pointing out a mistake in our synchronous algorithm. The authors are indebted to the anonymous referees who have given various valuable comments in improving the readability of the paper, in particular, the simplified example with time complexity $\Omega(n \log^* n)$ for the asynchronous algorithm.

REFERENCES

[1] B. AWERBUCH, *Hierarchical routing with small memory per node*, unpublished manuscript, October 1985.

[2] ———, *Complexity of network synchronization*, J. Assoc. Comput. Mach., 32 (1985), pp. 804–823.

[3] B. AWERBUCH AND R. G. GALLAGER, *Distributed breadth-first-search algorithms*, in Proc. 26th Annual IEEE Symposium on Foundations of Computer Sciences, IEEE Computer Society, Washington, DC, 1985, pp. 250–256.

[4] B. AWERBUCH AND S. MICALI, *Dynamic deadlock resolution protocols*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986.

[5] B. AWERBUCH, *Optimal distributed algorithms for minimum weight spanning tree, counting, leader election and related problems*, in Proc. 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 230–240.

[6] ———, *Linear time algorithm for minimum network partition*, TR Memo MIT/LCS/TM-350, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, March 1988.

[7] O. BORUVKA, *O jistém problému minimálnim*, Práce Mor. Přirodověd. Spol. v Brně (Acta Societ. Scient. Natur. Moravicae), 3 (1926), pp. 37–58.

[8] ———, *Přispěvek k řešeni otázky ekonomické stavby elektrovodnich siti*, Elektrotechnický obzor, 15 (1926), pp. 153–154.

[9] M. Y. CHAN, *Election and symmetry breaking in synchronous general networks*, TR-B7-86, University of Hong Kong, Hong Kong, July 1986.

[10] E. GAFNI AND Y. AFEK, *Time and message bounds for election in synchronous and asynchronous complete networks*, in Proc. 4th Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1985, pp. 186–195.

[11] E. GAFNI, *Improvements in the time complexity of two message-optimal electron algorithm*, in Proc. 4th Annual ACM Symposium on Principles of Distributed Computing, Association for Computing Machinery, New York, 1985, pp. 175–185.

[12] R. G. GALLAGER, *Finding a leader in a network with $O(E + N \log N)$ messages*, Massachusetts Institute of Technology, Cambridge, MA, 1977.

[13] R. G. GALLAGER, P. A. HUMBLET, AND P. M. SPIRA, *A distributed algorithm for minimum-weight spanning trees*, ACM Trans. Programming Languages and Systems, 5 (1983), pp. 66–77.

[14] R. L. GRAHAM AND P. HELL, *On the history of the minimum spanning tree problem*, Ann. Hist. Comput., 7 (1985), pp. 43–57.

[15] P. A. HUMBLET, *A distributed algorithm for minimum wieght directed spanning trees*, IEEE Trans. Comm., 31 (1983), pp. 756–762.

[16] J. PACHL, E. KORACH, AND D. ROTEM, *Lower bounds for distributed maximum-finding algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 905–918.

[17] D. PELEG AND E. UPFAL, *A tradeoff between size and efficiency for routing tables*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 43–52.

[18] P. M. SPIRA, *Communication complexity of distributed minimum spanning tree algorithms*, in Proc. 2nd Berkeley Conference on Distributed Data Management Computing Networks, Berkeley, CA, June 1977.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 627-643, August 1990

003

ON THE ANALYSIS OF SYNCHRONOUS COMPUTING SYSTEMS*

J. M. JOVER†‡, T. KAILATH†, H. LEV-ARI†, AND S. K. RAO†‡

Abstract. This paper is concerned with the analysis of synchronous, special purpose, multiple-processor systems (including, e.g., systolic arrays). The analysis problem is that of determining the algorithm executed by the system. There has been some prior work in this area, especially by Melhem and Rheinboldt [*SIAM J. Comput.*, 13 (1984), pp. 541-565], who were the first to obtain a general solution. The approach used here is different and apparently simpler. By combining ideas well known in system theory with certain graph-theoretical concepts, a simple procedure for recovering, within a natural equivalence, the iterative algorithm executed by a given special purpose synchronous computing array is obtained. The solution is based on reversing (modulo equivalence) the process by which an iterative algorithm is translated into a logical circuit.

Key words. dedicated multiple-processor systems, analysis of algorithm/logical graphs

AMS(MOS) subject classifications. primary 94A20; secondary 68A10

1. Introduction. In this paper we are concerned with the *analysis problem* of determining the algorithm executed by a given synchronous, special-purpose, multiple-processor array. In the analysis problem we are given the topology of the network, the function performed by each processor (including timing information), and the input data streams, and wish to determine the algorithm performed by the array. The problem arises because such arrays (or architectures) are often designed heuristically. Several formulations have been suggested in the computer science literature to solve a simpler related problem, often called *verification*, in which we want to check that a *given* algorithm is indeed implemented by the architecture.

Previous work on verification is due to Kung and Leiserson [9]; Chen and Mead [1], Lev-Ari [14], Kung and Lin [10], Kuo, Lévy, and Musicus [12], and Tidén [17]. The methods known so far tend to be somewhat involved and of limited generality. More general results, encompassing both verification and analysis, appear in a paper of Melhem and Rheinboldt [15], but their procedure is still quite involved. The main contribution of this paper is a new approach that leads to an apparently much simpler general solution.

The key to our approach is that we view the analysis problem as part of a cycle: starting from an algorithm, we design (or synthesize) a physical circuit; then we can complete the cycle (i.e., solve the analysis problem) by properly retracing our steps to recover the original algorithm (or rather one that it is equivalent to it in an appropriate sense). The first step in this cycle is to represent a given iterative algorithm, i.e., a set of relations between sequences of data, by a so-called *signal flow graph* (SFG), which shows the interconnections between blocks that perform ideal mathematical operations (i.e., take zero time to compute). The next step in the cycle is to modify the chosen SFG to obtain a *logical circuit* (i.e., a hardware implementation with physical modules that compute the same functions as the blocks in the SFG, but in nonzero time and

* Received by the editors June 30, 1986; accepted for publication (in revised form) September 6, 1989. This work was supported in part by the U.S. Army Research Office under contract DAAG29-83-K-0028; by the Air Force Office of Scientific Research, Air Force Systems Command under contract AF-83-0228; by the SDI/IST Program managed by the Army Research Office under contract DAAL03-87-K-0033; the Department of the Navy, Office of Naval Research under contract N00014-86-K-0726; by the Department of the Navy, Office of Naval Research under contract N00014-85-K-0612.

† Information Systems Laboratory, Stanford University, Stanford, California.

‡ Present address, AT&T Bell Laboratories, Holmdel, New Jersey 07733.

with some explicit delays). Our point is that we can solve the *analysis problem* by reversing the above path: modify the logical circuit to obtain a SFG, and thereby an associated algorithm, equivalent to the one we started with.

Therefore to solve the analysis problem it is helpful to understand the design phase, which is our first object of attention in this paper. The whole cycle will be explored in some detail using a simple example from linear system theory; in fact this example was the one that helped us to understand the analysis problem in a context more familiar to us, since for linear systems the design and analysis problems are well understood and there are well-established techniques, such as z-transforms and block-diagram-manipulations, to solve them (see, e.g., Kailath [7]).

2. Algorithms, SFGs, and algorithm graphs. Consider the iterative expression

$$y(k) = b_1u(k-1) - a_1y(k-1) + b_2u(k-2) - a_2y(k-2) + b_3u(k-3) - a_3y(k-3),$$

which describes the relation between two sequences, $u(\bullet)$ and $y(\bullet)$, that constitute a so-called linear filter. This filter produces a sequence of output values $\{y(k)\}$, given, at each k, certain past values of $y(\bullet)$ and of an *input sequence* $u(\bullet)$. Representations of this algorithm using simple building blocks—adders, multipliers, and separators (or index-shifting blocks)—can be set up in *many ways* (see, e.g., Kailath [7, Chap. 2]). One of these, the so-called observer form, is shown as a *signal flow graph* (SFG) in Fig. 1, where we have used a convention (arising from the use of what are called z-transforms) common in system theory of labeling the separator blocks by the symbol z^{-1}.

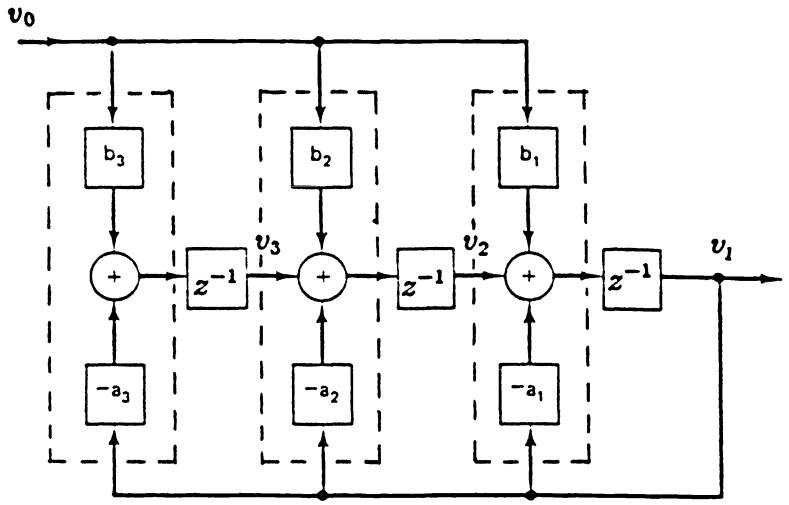


FIG. 1. *Observer canonical form (modified from* [7, *p.* 43]). *We define nodes* $\{v_0, v_1, v_2, v_3\}$ *as shown: one at the input point, and the others at the outputs of the* z^{-1} *blocks.*

Any signal-flow-graph is a *network of connected blocks.* The interconnecting wires propagate *sequences* of data elements, which we shall call *variables.* The points at which variables appear will be called *nodes.* Thus, for instance, the variable $x_1(k)$ denotes the sequence of data elements that appears (for $k = 0, 1, \cdots$) at the output (i.e., at the node v_1) of the linear filter in Fig. 1. The *processors* (=blocks) of a SFG transform one or several input variables into a single output variable. In general, this transformation need not be linear. The set of all variables and all the transformations determined by the processors constitutes the *iterative algorithm* performed by the SFG.

Now, with the important convention that arithmetic operations are *instantaneous*, i.e., that the input and output quantities have the same indices, while the separators (or z^{-1} blocks) shift the indices by unity, we can write the following (so-called "state[1]") equations:

$$(1)\qquad \begin{cases} x_1(k) = b_1u(k-1) - a_1x_1(k-1) + x_2(k-1), \\ x_2(k) = b_2u(k-1) - a_2x_1(k-1) + x_3(k-1), \\ x_3(k) = b_3u(k-1) - a_3x_1(k-1), \\ y(k) = x_1(k). \end{cases}$$

Note that these state equations actually represent an *aggregated* SFG, corresponding to the modules described by broken lines in Fig. 1. Conversely, it is easy to see how to draw this aggregated SFG from (1) (Fig. 2).

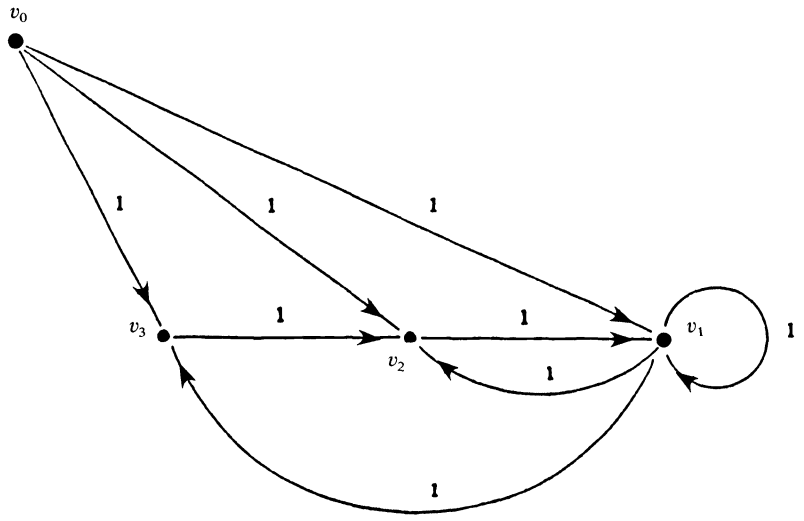


FIG. 2. *Algorithm graph for the aggregated observer canonical form.*

This representation, which we call the *algorithm graph*, has one vertex for every variable in (1) (with the exception of the rather trivial equation $y(k) = x_1(k)$, which we choose to ignore), and a directed arc from vertex v_i to vertex v_j whenever $x_j(\bullet)$ is a function of $x_i(\bullet)$. A multivariable function $f_j(\bullet)$ is associated with every vertex; its arguments are the variables $x_i(\bullet)$ whose vertices v_i are connected by (directed) arcs to v_j. For instance, the function associated with the vertex v_2 is $f_2(u, x_1, x_2, x_3) = b_2u - a_2x_1 + x_3$. The index displacement information, which is not present in the functions $f_j(\bullet)$ is displayed by the weights of the arcs (incoming into v_j) in the algorithm graph. Thus, for instance, the weight of the arc from v_1 to v_2 is one, indicating that $x_1(k)$ must be delayed by one time unit before it can be used to compute $x_2(k)$ via $f_2(\bullet)$.

To summarize the above discussion, we can say that generally the first step in obtaining a physical implementation is to start with an input-output description and

[1] The values $\{x_1(k), x_2(k), x_3(k)\}$ describe the "state" of the system at time k, in the sense that knowing them and $\{u(l), l \geqq k\}$ we can compute $\{y(l), l \geqq k\}$ irrespective of the prior values of the $x_i(\bullet)$, i.e., of $\{x_1(j), x_2(j), x_3(k), j < k\}$.

then to convert it, perhaps via the intermediate step of constructing some (aggregated) SFG representation, into an *iterative algorithm*, which is a set of equations of the form

$$(2)\qquad \begin{cases} x_1(k) = f_1\{x_1(k-s_{1,1}), x_2(k-s_{2,1}), \cdots, x_n(k-s_{n,1})\} \\ x_2(k) = f_2\{x_1(k-s_{1,2}), x_2(k-s_{2,2}), \cdots, x_n(k-s_{n,2})\} \\ \quad\vdots \\ x_n(k) = f_n\{x_1(k-s_{1,n}), x_2(k-s_{2,n}), \cdots, x_n(k-s_{n,n})\} \end{cases}$$

where k is the index of iteration, and $s_{i,j}$ are known as the *index displacements*. We emphasize that this conversion procedure is highly nonunique: there are many algorithms that can implement a given input-output map. However, there is a one-to-one correspondence between (2) and the corresponding algorithm graph, which has a function $f_j(\bullet)$ associated with every vertex v_j, and an arc from v_i to v_j with weight $s_{i,j}$ (in general, we allow multiple arcs from v_i to v_j with weights $s_{ij}^{(1)}, \cdots, s_{ij}^{(\mu_{ij})}$). Thus, an iterative algorithm is completely characterized by the quadruplet $\{\mathcal{I}, \mathcal{X}, \mathcal{F}, \mathcal{S}\}$, where:

- $\mathcal{I}$ is the *index space* (i.e., the set of all values of k for which (2) holds). Most often, it is the set of nonnegative integers, viz., $\mathcal{I} = \{k;\ 0 \leqq k < \infty\}$.
- $\mathcal{X}$ is the *variable set*, i.e., $\mathcal{X} = \{x_i(\bullet);\ 1 \leqq i \leqq n\}$.
- $\mathcal{F}$ is the *function set*, i.e., $\mathcal{F} = \{f_i(\bullet);\ 1 \leqq i \leqq n\}$.
- $\mathcal{S}$ is the *index displacement set*, i.e., $\mathcal{S} = \{s_{i,j}^{(r)};\ 1 \leqq i, j \leqq n,\ 1 \leqq r \leqq \mu_{i,j}\}$.

The nonnegative integer $\mu_{i,j}$ is called the *multiplicity* of the arc from v_i to v_j: if there is no such arc then $\mu_{i,j} = 0$, and otherwise $\mu_{i,j}$ equals the number of arcs from v_i to v_j. The total number of arcs in the algorithm graph is, therefore, $M := \sum_{i,j=1}^{n} \mu_{i,j}$.

For the analysis problem, we need only to be concerned with the dependence relations of iterative algorithms. That is, we will ignore in the sequel the explicit nature of the functional relations $\mathcal{F}$, and we will focus only on the information conveyed by the index displacement set $\mathcal{S}$. A convenient and concise way to summarize this information is the *algorithm matrix*

$$(3)\qquad G = \begin{pmatrix} C \\ D \end{pmatrix}$$

whose dimensions are $(n+1) \times (\Sigma \mu_{i,j})$. The last row of this matrix, which we denote by D, contains all the index displacements $s_{i,j}^{(r)}$ (the order of the columns of G is arbitrary). The first n rows of the algorithm matrix G form a *connection matrix*: the $n \times 1$ (column) vector in C above the element $s_{i,j}^{(r)}$ of D has a $+1$ element in the ith position, a -1 element in the jth position and zeros elsewhere (when $i = j$ the entire column of C consists of zeros). For instance the algorithm matrix corresponding to the algorithm graph of Fig. 2 is

$$(4)\qquad G = \left(\begin{array}{cccccccc} 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 0 & 1 & 1 & -1 & 0 \\ 0 & -1 & 0 & 0 & -1 & 0 & 1 & -1 \\ 0 & 0 & -1 & 0 & 0 & -1 & 0 & 1 \\ \hline 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \end{array}\right)$$

where the ordering of the eight columns corresponds to the following ordering of the arcs: (1) $v_0 \to v_1$, (2) $v_0 \to v_2$, (3) $v_0 \to v_3$, (4) $v_1 \to v_1$, (5) $v_1 \to v_2$, (6) $v_1 \to v_3$, (7) $v_2 \to v_1$, (8) $v_3 \to v_2$. Since the algorithm matrix G completely determines the algorithm graph, and vice versa, we will use the symbol G to denote both the matrix *and* the graph.

3. Logical circuits and logical graphs. SFGs are not truly "physical" implementations of mathematical algorithms such as (1) or (2), because in any physical hardware implementation, the arithmetic operations will *not* be instantaneous. One way to accommodate these physical constraints (and to interpret the SFG as a physical system) is by taking the iteration interval (i.e., the physical time separation between sequence elements) to be very large, so that the arithmetic operations in each computing module will all be completed before the next iteration begins, i.e., before the next data sample is entered into the system. A more efficient procedure, likely to result in smaller iteration intervals, is to determine a "schedule" of the times at which each operation should be performed, as explained next.

We will confine ourselves to synchronous digital implementations, in which we have an underlying clock, whose period will be taken as the basic time unit. Then the time required for additions and multiplications (or other arithmetic operations) will be measured as integral multiples of clock cycles. We will not concern ourselves with the details of what happens *within* any particular clock cycle.

The main goal of the scheduling procedure is to determine an appropriate *iteration interval*, i.e., the physical time (measured in clock cycles) between two consecutive data at any point in the system (this will be the same at all points in a synchronous system), and any additional delays required, called *shimming delays*, that may have to be added to the processing and transmission delays of the system to ensure that the proper elements in the various sequences are interacting correctly.

Several algorithms for scheduling have been developed. Here we use the ideas of Jagadish et al. [4] (see also Jagadish [5] and Rao [16]) to determine a scheduling for the observer canonical form. Suppose that multiplication (and data transfer) takes seven clock cycles, addition (and transfer) takes three clock cycles, and a pure transfer of data along an interconnecting wire takes one clock cycle. Applying the scheduling procedure of Jagadish et al. [4] will yield many (equivalent) possible physical implementations, one of which is shown in Fig. 3: the blocks represent hardware components with computational delays as assumed above, and there are additional

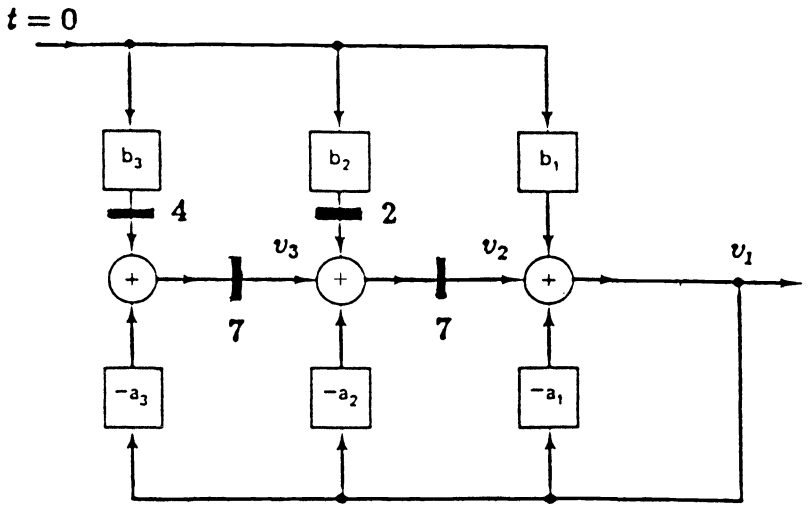


FIG. 3. *Logical circuit for the observer canonical form*: (a) *timing for modules*, (b) *logical circuit. The square multiplication blocks have delay seven clock cycles, adders have delay three clock cycles, and each data transfer takes one clock cycle.*

(shimming) delays whose value (in multiples of clock cycles) is indicated next to them. Note that the delays corresponding to multiplication, addition, and data transfer are not explicitly noted. The iteration interval for this circuit turns out to be eleven clock cycles. Digital designers usually call such a figure a logical (circuit) diagram.[2] It should be emphasized that the scheduling procedure is highly nonunique and, therefore, that several different logical circuit diagrams can be associated with a given SFG. However, this multiplicity of choices does not really concern us; we can start with any one of them and show how to recover (up to a certain equivalence) the original SFG.

3.1. Logical graphs. For many purposes, especially timing analysis, it is convenient to redraw the logical circuit diagram in a stripped-down form called a *logical graph* (see Fig. 4). As the algorithm graph, the logical graph has one vertex for each variable, and its edges represent the dependencies between the variables. However, the weight of its $v_i \to v_j$ edges represents the total *computational and propagation delay* $d_{i,j}$ for this path, rather than the index displacement $s_{i,j}$. For example, from v_2 to v_1 we have an edge with weight $d_{2,1} = 3$ (corresponding to a single addition), a self-loop from v_1 to itself with weight $d_{1,1} = 1+7+3 = 11$ (corresponding to a data transfer, a multiplication, and an addition), a path from v_1 to v_2 with weight $d_{1,2} = 1+1+7+3+7 = 19$, and so on.

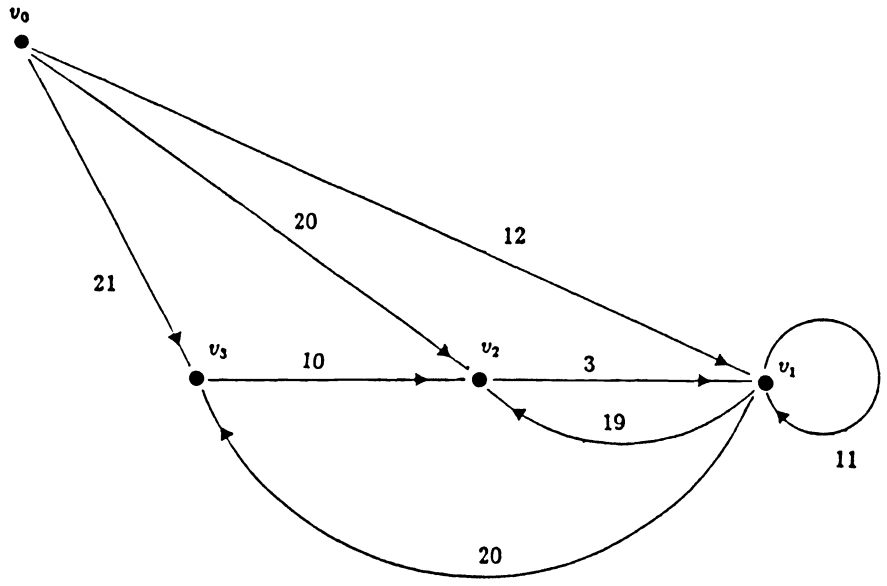


FIG. 4. *Logical graph for the aggregated observer canonical form.*

Given the functional information about the function associated with each vertex, we can recover from the logical graph the following set of equations:

$$
(5)\qquad \begin{cases} \tilde{x}_1(l) = b_1\tilde{u}(l-12) - a_1\tilde{x}_1(l-11) + \tilde{x}_2(l-3), \\ \tilde{x}_2(l) = b_2\tilde{u}(l-20) - a_2\tilde{x}_1(l-19) + \tilde{x}_3(l-10), \\ \tilde{x}_3(l) = b_3\tilde{u}(l-21) - a_3\tilde{x}_1(l-20), \\ \tilde{y}(l) = \tilde{x}_1(l). \end{cases}
$$

[2] The adjective "logical" arises from the fact that the hardware is based on so-called "logical" components obeying the rules of Boolean logic (algebra).

Note that we needed to introduce in (5) a new index l and that these equations do not resemble the original equations (1), since there seems to be no simple relationship between the $d_{i,j}$ and the index displacements $s_{i,j}$. Finding these index dependencies (i.e., the index displacements $s_{i,j}$ in (2)) will solve the analysis problem.

Actually, (5) already provides one possible solution to the analysis problem. However, this solution is highly inefficient: computations do not take place at every index point (clock cycle) l but rather at time instants separated by several clock cycles. Thus, for instance, inputs are introduced at the vertex v_0 only at the time instants $l = 11k$, i.e., $\tilde{u}(l)$ is *undefined* for $l \neq 11k$. In order to find a solution that resembles (1) more closely, we should try to reverse the synthesis procedure (effectively, the scheduling procedure) that generates the logical circuit implementation of a given iterative algorithm.

3.2. Algorithm graphs versus logical graphs. We can get some more insight into doing this reversal by noting that the scheduling method of Jagadish et al. [4] is a somewhat simpler method for achieving a result that was previously obtained via so-called multirate *systolization procedures* (see, e.g., Kung [11] or Ullman [18]). Such procedures consist of two basic steps:

- rescaling the index, viz., $l := \delta k$,
- moving delays (separator blocks) in the network (via cutset transformations or other methods), which amounts to individually shifting the index for each variable $x_i(\bullet)$, viz., $l := k - \lambda_i$.

The combined effect of both types of transformations is to replace $x_i(k)$ by $\tilde{x}_i(l)$ where

$$\tilde{x}_i(\delta k + \lambda_i) := x_i(k) \tag{6}$$

and $\tilde{x}_i(l)$ is undefined for $l \neq \delta k + \lambda_i$. Consequently, a dependence of the form $x_i(k - s_{i,j}) \to x_j(k)$ now becomes $\tilde{x}_i[\delta(k - s_{i,j}) + \lambda_i] \to \tilde{x}_j(\delta k + \lambda_j)$ and, therefore, the new index displacements (for the index l) are

$$d_{i,j}^{(r)} = (\lambda_j - \lambda_i) + \delta s_{i,j}^{(r)}. \tag{7}$$

This is the basic equation that must be satisfied by every logical graph $\tilde{G}$ that implements a given algorithm graph G.

Another more intuitive way to obtain the same relation is by analyzing the way synchronous systems work. In synchronous systems the time between two consecutive elements in any sequence is constant (and equal to the iteration interval, δ). If in addition we assume that such systems are time-invariant, as we do in logical graphs, all the computations (at the vertices) involve data arriving at some multiple of the iteration interval. Consider, for example, the graph $\tilde{G}$ for the observer form (Fig. 4). Apply the first element $u(1)$ of an input sequence $u(\bullet)$ at time $\lambda_0 = 0$; by definition, the rest of the elements will be generated every δ clock cycles. Let us denote by λ_i the time instant at which the vertex v_i generates the output $x_i(1)$. Recall that $d_{i,j}^{(r)}$ (respectively, $s_{i,j}^{(r)}$) is the physical delay in the logical graph $\tilde{G}$ (respectively, the index displacement in the algorithm graph G) along a directed path connecting the vertex v_i to the vertex v_j. Since v_i generates $x_i(1)$ at time λ_i, $x_i(1)$ arrives at vertex v_j at the time $\lambda_i + d_{i,j}^{(r)}$, for $r = 1, 2, \cdots, \mu_{i,j}$. Clearly, we cannot equate λ_j with $\lambda_i + d_{i,j}^{(r)}$, because λ_j will depend on the number of separators in the paths from v_i to v_j, which in turn will fix the actual iterations in which the input $x_i(1)$ is operated on at vertex v_j. In our case, a path from v_i to v_j has $s_{i,j}^{(r)}$ separators and therefore, the vertex v_j will associate

the input $x_i(1)$ with the $(1+s_{i,j}^{(r)})$th iteration rather than with the first iteration. Consequently, v_j will generate $x_j(1)$ at time $\lambda_j = \lambda_i + d_{i,j}^{(r)} - s_{i,j}^{(r)}\delta$ where δ denotes the iteration interval. It follows that, for *every path* from v_i to v_j, $d_{i,j}^{(r)} = (\lambda_j - \lambda_i) + s_{i,j}^{(r)}\delta$, which is precisely the relation (7). The constants λ_i determine a *schedule* for G in the sense that $x_i(k)$ is mapped into $\tilde{x}_i[\delta(k-1)+\lambda_i]$.

Returning now to the general discussion, we note that a transformation of the index space, linear (such as (6)) or otherwise, does not affect the actual computations that produce the variable $x_i(k)$, nor does it affect the precedence (=dependence) relations among those variables. Thus both (1) and (5) can be thought of as representations of the same algorithm, and the relation (7) as an *equivalence transformation.* This interpretation is underscored by the observation that (7) can be rewritten in the form

$$\begin{pmatrix} C \\ \tilde{D} \end{pmatrix} = \begin{pmatrix} I & 0 \\ \Lambda & \delta \end{pmatrix} \begin{pmatrix} C \\ D \end{pmatrix}, \qquad \Lambda := [\lambda_1 \lambda_2 \cdots \lambda_n] \tag{8}$$

where the elements of the row vector $\tilde{D} = \{d_{i,j}^{(r)}\}$ are ordered in exactly the same manner as those of $D = \{s_{i,j}^{(r)}\}$. Since $\delta \neq 0$ the transformation matrix $\left(\begin{smallmatrix} I & 0 \\ \Lambda & \delta \end{smallmatrix}\right)$ is nonsingular and (8) qualifies as an equivalence relation.

There is, however, one important difference between the equivalent representations (1) and (5). The former is executed at every index point (i.e., its iteration interval is one), while the latter is executed only at a subset of all index points (i.e., its iteration interval is greater than one). We will call a representation *compact* when its iteration interval is unity and *noncompact* otherwise.

3.3. The analysis problem. In this language, we see that the synthesis (scheduling) problem is to transform a compact representation given by an algorithm graph into a (possibly noncompact) equivalent representation given by a logical graph with the property that $d_{i,j}^{(r)} \geqq h_i$ for all i, j, r, where h_i is the time required to evaluate $x_i(k) = f_i(\cdots)$.

The analysis problem is, in essence, the converse to the synthesis problem: to transform a noncompact representation (i.e., a logical graph) into a compact equivalent (i.e., an algorithm graph). This problem does not possess a unique solution, as every noncompact representation has many compact equivalents. It would seem, therefore, that the only way to solve this problem is by an exhaustive search in the space of all equivalent representations until a compact one has been found. It turns out, however, that by extracting some additional information from the logical circuit itself we can construct a compact equivalent without any search whatsoever. A systematic procedure for carrying out such a construction is described in the next section.

The first step in solving the analysis problem is to determine the iteration interval of a given logical circuit. The iteration interval is determined by the computational delay around loops in the logical circuit, and equals, in fact, the *greatest common divisor* (gcd) of the computational delay around all loops (see, e.g., Jagadish et al. [4]). We remark here that the computational delay around a loop is obtained by adding up the delay of all arcs that point in one direction (say clockwise), and subtracting the delays of all arcs that point in the opposite direction. Since every loop in a graph is a combination of *fundamental loops* (i.e., loops determined by a spanning tree), the iteration interval can be computed as the gcd of the computational delays around fundamental loops alone. We will denote the gcd of all fundamental loops in an algorithm/logical graph G by $\eta(G)$. It follows that if

$$\tilde{G} = \begin{pmatrix} C \\ \tilde{D} \end{pmatrix} = \begin{pmatrix} I & 0 \\ \Lambda & \delta \end{pmatrix} \begin{pmatrix} C \\ D \end{pmatrix},$$

then $\eta(\tilde{G}) = \delta\eta(G)$. To prove this observe that G and $\tilde{G}$ have the same matrix C (i.e., they have the same topology), but a different set of arc weights. A loop in G (or in $\tilde{G}$) corresponds to an integer-valued column vector q such that $Cq = 0$, and its weight in G is given by Dq. The weight of the same loop in $\tilde{G}$ is given by

$$\tilde{D}q = [\Lambda\delta]\begin{pmatrix} C \\ D \end{pmatrix} q = \delta Dq$$

and since the same holds, in particular, for all *fundamental loops*, we conclude that $\eta(\tilde{G}) = \delta\eta(G)$, as stated. Moreover, every algorithm/logical graph with $\eta(G) > 1$ (i.e., noncompact) can be transformed into an equivalent $\breve{G}$ with $\eta(\breve{G}) = 1$ (i.e., compact), which corresponds to the shortest possible iteration interval. One way to accomplish this transformation is by using the analysis procedure that we will present in the following section.

Our last observation for this section is that the constraints λ_i in the fundamental equation (7) need only to be specified modulo δ. This is so because adding $m_i\delta$ to λ_i has the same effect as replacing the original algorithm graph G by an *equivalent* graph $\breve{G}$, where

$$\breve{G} = \begin{pmatrix} I & 0 \\ M & 1 \end{pmatrix} G, \qquad M := [m_1 m_2 \cdots m_n].$$

4. Analysis procedure. We now present an analysis procedure that constructs a set of λ_i and an algorithm graph G with *nonnegative* index displacements $s_{i,j}^{(r)}$ for any given logical graph $\tilde{G}$. First, we assume that the logical graph has at least one vertex with no incoming edges, which we will call the *input*. If there are several input vertices we also assume that the values of the constants λ_i are known for these vertices. This amounts to the assumption that we know the relative *skewing* (i.e., alignment) of the input sequences to the algorithm. Next we *add* a preferred input vertex, which we will call the *root*, and connect it with arcs to all the original input vertices assigning a delay λ_i to the arc that connects the root v_0 to the input vertex v_i. We also assume that the resulting *extended logical graph* $\tilde{G}_{\text{ext}}$ is *root-connected*, i.e., it has at least one directed path from the root to every vertex of the graph. Finally, we form a *rooted tree* consisting of the shortest (=minimum-delay) paths from the input vertex v_0 to every vertex of $\tilde{G}_{\text{ext}}$. By setting $s_{i,j} = 0$ for the edges contained in the rooted tree, we can determine λ_i for all vertices ($l_0 = 0$ for the root vertex) and we can use (7) to compute the index displacements $s_{i,j}$ for the edges that are not in the rooted tree. These edges, which we will call the *links*, also serve to determine a set of fundamental loops.

Our procedure can be formally summarized as follows.

ANALYSIS PROCEDURE to determine a compact equivalent to a given root-connected extended logical graph $\tilde{G}_{\text{ext}}$:

1. Form a spanning tree with the minimum-delay path from v_0 to every vertex in $\tilde{G}_{\text{ext}}$. Set λ_i equal to the length of the minimum-delay path from v_0 to v_i. Set $\hat{s}_{i,j} := 0$ for every edge in the tree.
2. For every link of the minimum-delay tree (i.e., every edge not in the tree), including all self loops, compute $\hat{d}_{i,j}^{(r)} = \lambda_i - \lambda_j + d_{i,j}^{(r)}$.
3. Set $\hat{s}_{i,j}^{(r)} := \hat{d}_{i,j}^{(r)}/\delta$ for every link of the minimum-delay tree, where $\delta := \gcd\{\hat{d}_{i,j}^{(r)}\}$.

This procedure constructs an algorithm graph $\hat{G}$ with (integer) nonnegative index displacement $\hat{s}_{i,j}^{(r)}$, which is a compact equivalent of the given extended logical graph $\tilde{G}_{\text{ext}}$. Removal of the root vertex v_0 results in a compact equivalent of the original (nonextended) logical graph $\tilde{G}$.

There are several algorithms for implementing step 1 of the procedure (shortest-path tree), such as those in Dantzig [2]. For a comprehensive presentation of these algorithms see, e.g., Even [3].

As a simple example, consider the observer form of (5). Note that an extension (step 1 of our procedure) is not required, because there is only one input vertex v_0, for which we choose $\lambda_0 = 0$. The corresponding shortest-path tree and the resulting index displacement are described in Fig. 5. From this graph we can easily write the following state equations:

$$(9) \qquad \begin{cases} \hat{x}_1(k) = b_1\hat{u}(k) - a_1\hat{x}_1(k-1) + \hat{x}_2(k-1), \\ \hat{x}_2(k) = b_2\hat{u}(k) - a_2\hat{x}_1(k-1) + \hat{x}_3(k-1), \\ \hat{x}_3(k) = b_3\hat{u}(k) - a_3\hat{x}_1(k-1). \end{cases}$$

We now observe that the algorithm graphs in Figs. 2 and 5 are different. The difference in the algorithms in (1) and (9) amounts only to shifts in the indices (we can change the indices for the sequence $\hat{u}(\bullet)$ in (9) from k to $k-1$ and obtain (1)) and therefore both algorithms are *equivalent and compact.* This is so because G and $\hat{G}$ are related by the equivalence transformation

$$\hat{G} = \begin{pmatrix} I & & 0 \\ 1\,0 \cdots 0 & & 1 \end{pmatrix} G$$

with the property $\eta(\hat{G}) = \eta(G) = 1$.

Returning to the general case, to establish the validity of our analysis procedure we need to prove the following statements: (i) $\hat{s}_{i,j}^{(r)} \geqq 0$, (ii) $\hat{G}$ is equivalent to G, and (iii) $\eta(\hat{G}) = 1$. First, observe that since we have formed a rooted shortest-path tree from v_0 to every vertex v_i, then for every link (v_i, v_j) of the tree $\lambda_i + d_{i,j}^{(r)} \geqq \lambda_j$ and, therefore, $\hat{d}_{i,j}^{(r)} = \lambda_i - \lambda_j + d_{i,j}^{(r)} \geqq 0$, which implies that $\hat{s}_{i,j}^{(r)} \geqq 0$ as well. Also, $\hat{s}_{i,j}^{(r)}$ is an integer, by construction. Next, note that the algorithm $\hat{G}$ recovered by our procedure satisfies the equation $\hat{\lambda}_j = \hat{\lambda}_i + d_{i,j}^{(r)} - \hat{s}_{i,j}^{(r)}\delta$, where $\delta := \eta(\tilde{G})$, while the original algorithm G, used to design the logical circuit $\tilde{G}$, satisfies the equation $\lambda_j = \lambda_i + d_{i,j}^{(r)} - s_{i,j}^{(r)}\delta$.

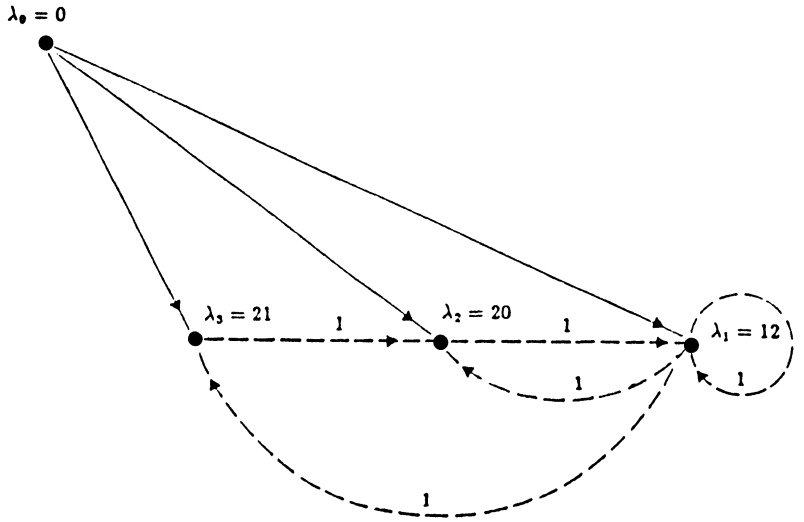


FIG. 5. *Shortest-path tree and the corresponding algorithm graph for the aggregated observer canonical form.*

Assuming all graphs in consideration have been extended, we can assume for the root vertex v_0 that $\lambda_0 = 0 = \hat{\lambda}_0$. Thus, in particular, $\hat{\lambda}_i - \lambda_i = (s_{i,0}^{(1)} - \hat{s}_{i,0}^{(1)})\delta = c_i\delta$, where $c_i := s_{i,0}^{(1)} - \hat{s}_{i,0}^{(1)}$, and, consequently, $\hat{s}_{i,j}^{(r)}\delta = \hat{\lambda}_i - \hat{\lambda}_j + d_{i,j}^{(r)} = \lambda_i - \lambda_j + d_{i,j}^{(r)} + (c_i - c_j)\delta = (s_{i,j}^{(r)} + c_i - c_j)\delta$. This means that

$$\hat{G} = \begin{pmatrix} I & 0 \\ c_1 \cdots c_n & 1 \end{pmatrix} G,$$

which establishes the equivalence between G and $\hat{G}$, with $\eta(G) = \eta(\hat{G})$. Finally, the relation between G and $\tilde{G}$ is

$$\tilde{G} = \begin{pmatrix} I & 0 \\ \lambda_1 \cdots \lambda_n & \delta \end{pmatrix} G,$$

which proves that $\eta(G) = \eta(\tilde{G})/\delta = 1$. This completes the proof of the statements (i)–(iii) and establishes the validity of our analysis procedure.

5. Examples. In this section we illustrate our procedure by analyzing some arrays previously verified or analyzed by other authors. Most of the authors limit their efforts to the Kung–Leiserson matrix-multiplication array; the most comprehensive effort was made by Melhem and Rheinboldt [15], who studied three additional examples: convolution, sorting, and back substitution. We analyze these three examples below; additional examples can be found in Jover [6]. Note that previous authors always assume that all the operations take one unit of time to perform (in our language, that the computational delays are unity for all the edges of the logical graph). In contrast, we can analyze multirate arrays, which involve different computational delays for different computations, since this does not complicate our analysis procedure at all.

5.1. Forward convolution. This array was developed by Kung and Leiserson [9]. Figure 6 depicts the array, the processors' functions, the logical graph $\tilde{G}$, and the shortest-path tree. The input sequence is applied at the vertex v_1, while a constant zero is applied at the vertex v_8. Since multiplication is assumed to require seven clock cycles, the input at the vertex v_8 has to be skewed by the same duration, so $\lambda_8 = 7$, assuming we choose $\lambda_1 = 0$. Addition is assumed to require three clock cycles.

Following the analysis procedure, we determine the shortest path between v_0 and the rest of the vertices. Figure 6(d) shows one choice for the shortest-path tree; links are indicated by dashed lines. Note that there are other choices for the shortest-path tree that have the same "path-length" λ_i: for instance, we can replace the arc $v_1 \to v_7$ by the arc $v_8 \to v_7$. Next, we compute $\hat{d}_{76} = 2 = \hat{d}_{65}$ and $\eta(\tilde{G}) = \gcd\{2, 2\} = 2$. Thus, the index displacements along the links are $\hat{s}_{76} = 1 = \hat{s}_{65}$ and $\hat{s}_{87} = 0$.

Finally, we can write the equations directly, viz.,

$$\begin{cases} x_2(k) = x_1(k), & x_7(k) = \omega_2 x_1(k) + x_8(k), \\ x_3(k) = x_2(k), & x_6(k) = \omega_1 x_2(k) + x_7(k-1), \\ x_4(k) = x_3(k), & x_5(k) = \omega_0 x_3(k) + x_6(k-1). \end{cases}$$

Since these equations are relatively simple we can eliminate intermediate variables to obtain the following input-output relation (where $y(k) = x_5(k)$ and $u(k) = x_1(k)$)

$$y(k) = \omega_0 u(k) + \omega_1 u(k-1) + \omega_2 u(k-2),$$

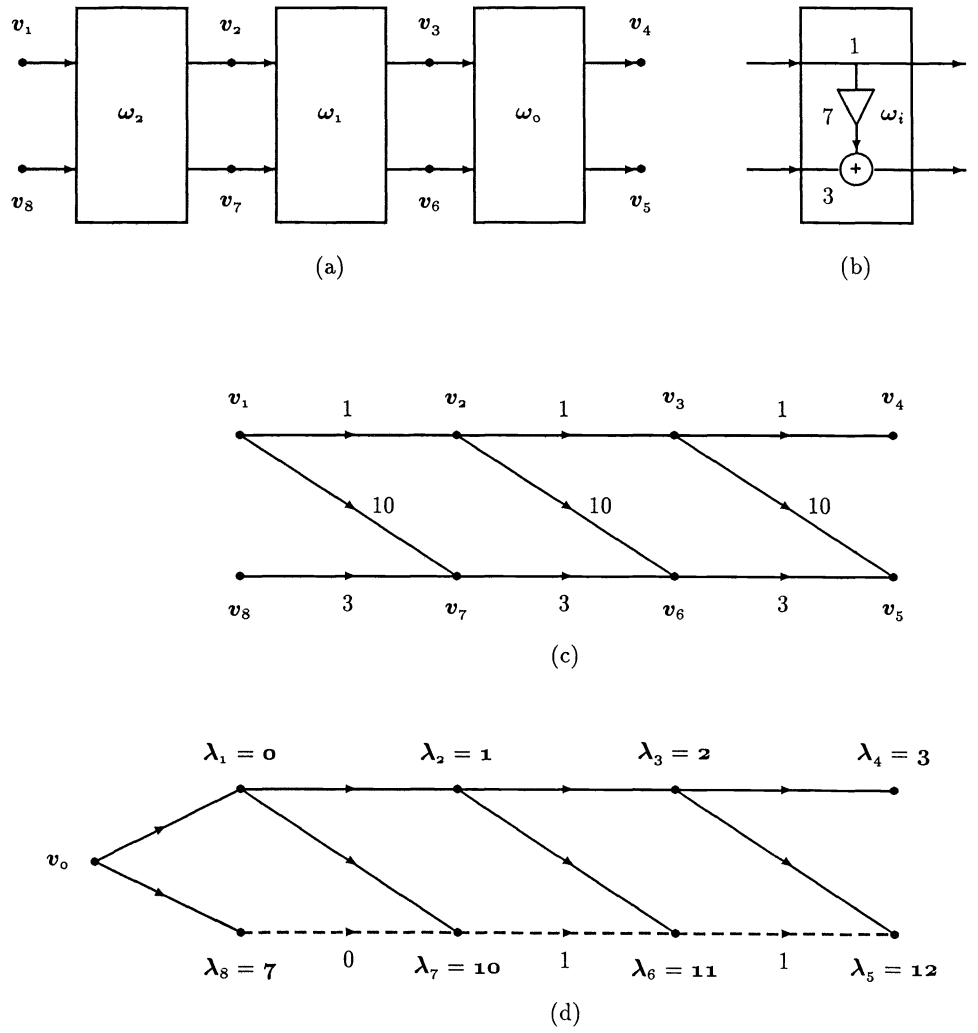


FIG. 6. *Forward convolution array*: (a) *system*, (b) *processor functions and timing*, (c) *logical graph* $\tilde{G}$, (d) *shortest-path tree.*

which is, clearly, a convolution. In general, elimination of internal variables may lead to complicated expressions for the input-output relation.

5.2 Back substitution. This array was developed by Kung and Leiserson [9] and verified by Melhem and Rheinboldt [15]. Figure 7 depicts the system, the processors' functions, the logical graph $\tilde{G}$, and a shortest-path tree. The root vertex is v_6; for the sake of clarity we do not show the arcs that are supposed to connect the root vertex to the remaining input vertices (v_1, v_2, v_3, v_4, and v_5), as would have been required by the analysis procedure of § 4. This time, we have associated a computational delay of 10 for each of the edges in the logical graph $\tilde{G}$; this choice corresponds to the usual one for single rate systolic arrays: all the outputs to a cell are produced at the same time even if some of the outputs may take less time to compute.

The input sequences are as follows: a constant zero is applied at the vertex v_6; the elements of a column vector b are applied at the vertex v_1, and the diagonals of

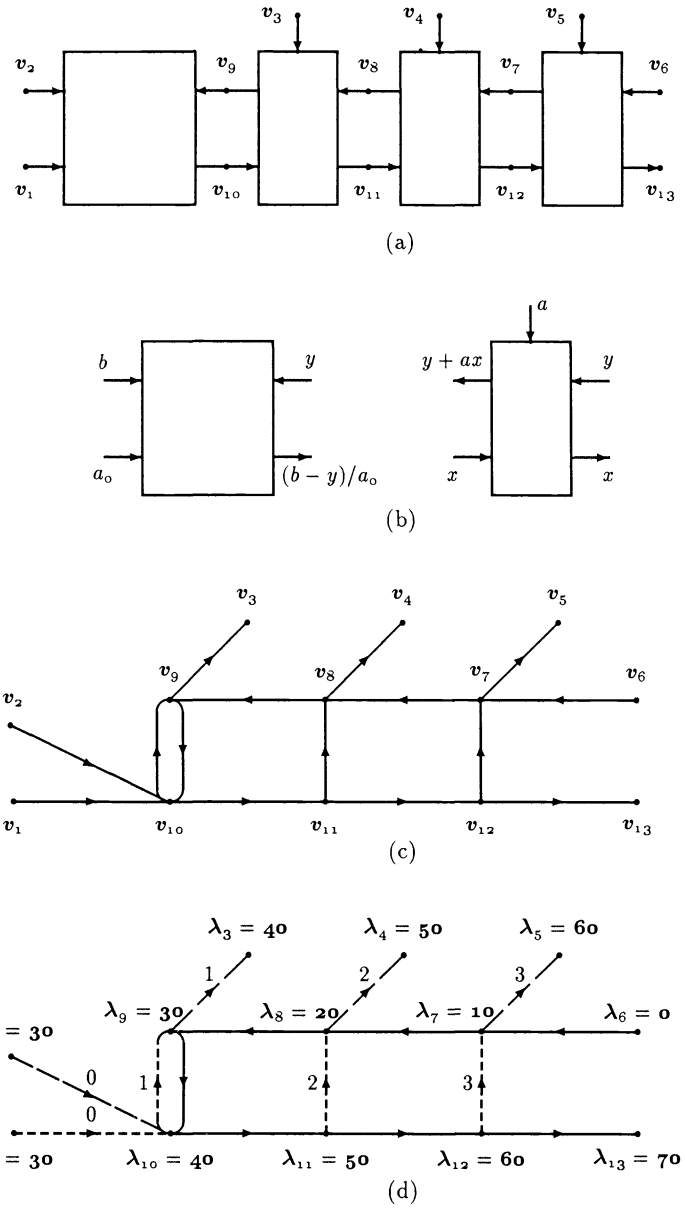


FIG. 7. *Back substitution array*: (a) *system*, (b) *processors' functions*, (c) *logical graph* $\tilde{G}$, *and* (d) *a shortest-path tree.*

a banded lower-triangular matrix A are applied at the vertices v_2 through v_5 (the main diagonal at v_2, the first subdiagonal at v_3, and so on). Thus, the input sequences are as follows:

$$\begin{cases} x_1 = \{b_1, b_2, b_3, \cdots\}, & x_2 = \{a_{11}, a_{22}, a_{33}, \cdots\}, \\ x_3 = \{a_{21}, a_{32}, a_{43}, \cdots\}, & x_4 = \{a_{31}, a_{42}, a_{53}, \cdots\}, \\ x_5 = \{a_{41}, a_{52}, a_{63}, \cdots\}. \end{cases}$$

The output is $y(k) = x_{13}(k)$. We are also given the times λ_i at which the first input of

each sequence is entered, as follows: $\lambda_1 = \lambda_2 = 30$, $\lambda_3 = 40$, $\lambda_4 = 50$, $\lambda_5 = 60$, $\lambda_6 = 0$. These times correspond to the weights in the edges connecting the root (not shown) and the input vertices.

Figure 7(d) shows the resulting index displacements in addition to the shortest-path tree. We computed them using $\eta(\tilde{G}) = \gcd\{20, 40, 60\} = 20$. From this figure we can write the following equations:

$$\begin{cases} x_7(k) = x_{12}(k-3) \cdot x_5(k-3) + x_6(k), \\ x_8(k) = x_{11}(k-2) \cdot x_4(k-2) + x_7(k), \\ x_9(k) = x_{10}(k-1) \cdot x_3(k-1) + x_8(j), \\ x_{10}(k) = (x_1(k) - x_9(k))/x_2(k), \\ x_{11}(k) = x_{10}(k), \quad x_{12}(k) = x_{11}(k), \quad x_{13}(k) = x_{12}(k). \end{cases}$$

Eliminating internal variables, we get

$$y(k) = \frac{x_1(k) - x_9(k)}{x_2(k)} = \frac{b_k - x_9(k)}{a_{kk}} = x_{10}(k) = x_{11}(k) = x_{12}(k)$$

where

$$\begin{aligned} x_9(k) &= y(k-1) \cdot x_3(k-1) + y(k-2) \cdot x_4(k-2) + y(k-3) \cdot x_5(k-3) + x_6(k) \\ &= y(k-1) \cdot a_{k,k-1} + y(k-2) \cdot a_{k,k-2} + y(k-3) \cdot a_{k,k-3}. \end{aligned}$$

These equations are the well-known method of solving a triangular system, called back substitution.

5.3. Sorting. This array was developed by Kung and first reported and verified by Melhem and Rheinboldt [15]. The system sorts a sequence of n real numbers, $u(\bullet) = x_1(\bullet) = \{u(1), u(2), \cdots, u(n)\}$ by using the linear array of $n-1$ processors depicted in Fig. 8, and produces the sorted output sequence, $y(\bullet) = x_8(\bullet) = \{y(1), y(2), \cdots, y(n)\}$. In this example, we assume all the computational delays to be equal and of value 10 since all the operations are well balanced and should take the same amount of time. We have only one source, so we take it directly as the root with $\lambda_1 = 0$. Figures 8(c) and 8(d) show the logical graph and the (unique) shortest-path tree with the corresponding index displacements. From them we can readily write the following equations:

$$\begin{cases} x_2(k) = \max\{x_1(k), x_7(k-1)\}, \\ x_3(k) = \max\{x_2(k), x_6(k-1)\}, \\ x_4(k) = \max\{x_3(k), x_5(k-1)\}, \\ x_5(k) = x_4(k), \\ x_6(k) = \min\{x_3(k), x_5(k-1)\}, \\ x_7(k) = \min\{x_2(k), x_6(k-1)\}, \\ x_8(k) = \min\{x_1(k), x_7(k-1)\}, \end{cases}$$

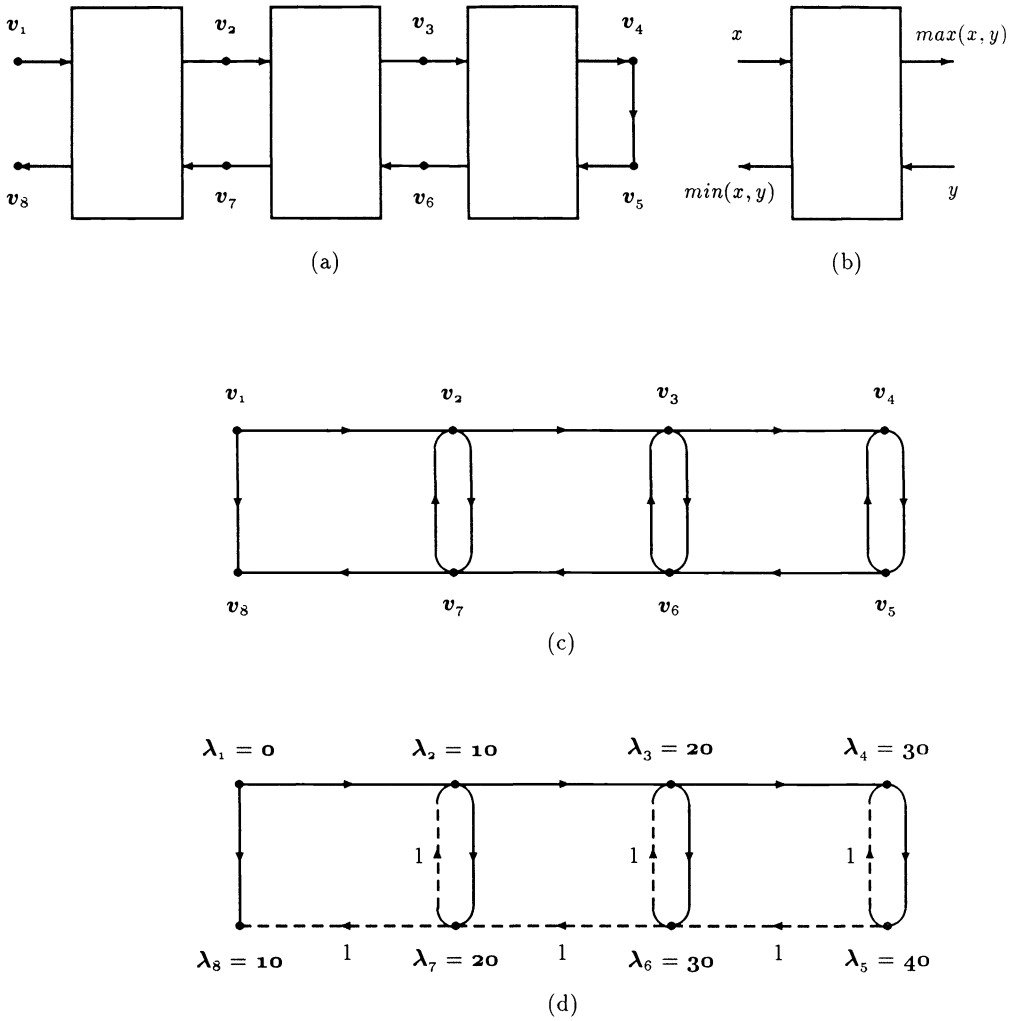


FIG. 8. *Sorting array*: (a) *system*, (b) *processors' functions*, (c) *logical graph* $\tilde{G}$, *and* (d) *the shortest-path tree.*

which can be rearranged as follows, after substituting $x_5(k) = x_4(k)$:

$$\begin{cases} x_2(k) = \max\{x_1(k), x_7(k-1)\}, \\ x_8(k) = \min\{x_1(k), x_7(k-1)\}, \\ x_3(k) = \max\{x_2(k), x_6(k-1)\}, \\ x_7(k) = \min\{x_2(k), x_6(k-1)\}, \\ x_4(k) = \max\{x_3(k), x_4(k-1)\}, \\ x_6(k) = \min\{x_3(k), x_4(k-1)\}. \end{cases}$$

These equations correspond to the so-called bubble sort (see Knuth [8]). Other types of sorting algorithms could be implemented (see, for instance, Rao [16] and Lang et al. [13]).

6. Concluding remarks. We have given a simple procedure for recovering (within a natural equivalence) the iterative algorithm executed by a given special purpose synchronous computing array. The solution is based on reversing (modulo equivalence) the process by which we can translate an iterative algorithm into a logical circuit.

A general theory for such conversion has recently been developed by Rao [16] and Jagadish [5] in their work on the analysis and synthesis of what they call *Regular Iterative Arrays* (RIAs). The synchronous circuits studied in this paper can be identified as a special class of RIAs (with a one-dimensional index space). Therefore, the algebraic techniques developed, in particular by Rao [16], can be applied to generalize the results of this paper to other classes of RIAs (with multidimensional index spaces). In particular, regularity of spatial structure, as in systolic arrays, can be exploited to reduce the study of such systems to that of a single module (see, e.g., Jover [6]).

Our analysis procedure is restricted to logical graphs in which every vertex is reachable from at least one source (input) vertex. It can be shown that logical graphs not possessing this property can be modified by adding edges from the root v_0 to some of the unreachable vertices, so that subsequently our analysis procedure can be applied (see Jover [6]).

REFERENCES

[1] M. C. CHEN AND C. A. MEAD, *Concurrent algorithms as space-time recursion equations*, in Proc. USC Workshop on VLSI and Modern Signal Processing, Los Angeles, CA, November 1982; VLSI and Modern Signal Processing, S. Y. Kung, H. J. Whitehouse, and T. Kailath, eds., Prentice-Hall, Englewoods Cliffs, NJ, 1985.

[2] G. B. DANTZIG, *On the shortest route through a network*, in Studies in Graph Theory, D. R. Fulkerson, ed., MAA Studies in Mathematics, Vol. 11, The Mathematical Association of America, Washington, DC, 1975.

[3] S. EVEN, *Graph Algorithms*, Computer Science Press, Rockville, MD, 1979.

[4] H. V. JAGADISH, R. G. MATHEWS, T. KAILATH, AND J. A. NEWKIRK, *A study of pipelining in computing arrays*, IEEE Trans. Comput., 35 (1985), pp. 431-440.

[5] H. V. JAGADISH, *Techniques for the design of parallel and pipelined* VLSI *systems for numerical computation*, Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA, December 1985.

[6] J. M. JOVER, *On the modeling and analysis of systolic and systolic-type arrays*, Ph.D. dissertation, Department of Electrical Engineering, Stanford University, Stanford, CA, December 1985.

[7] T. KAILATH, *Linear Systems*, Prentice-Hall, Englewood Cliffs, NJ, 1980.

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

[9] H. T. KUNG AND C. E. LEISERSON, *Systolic arrays (for* VLSI), in Sparse Matrix Proceedings 1978, I. S. Duff and G. W. Stewart, eds., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1979, pp. 256-282.

[10] H. T. KUNG AND W. T. LIN, *An algebra for* VLSI *algorithm design*, in Proc. Conference on Elliptic Problem Solvers, Monterey, CA, January 1983.

[11] S.-Y. KUNG, *On supercomputing with systolic/wavefront array processors*, Proc. IEEE, 72 (1984), pp. 867-884.

[12] C. J. KUO, B. C. LEVY, AND B. R. MUSICUS, *The specification and verification of systolic wave algorithms*, in 1984 USC Workshop on VLSI Signal Processing, Los Angeles, CA, November 1984.

[13] H. W. LANG, M. SCHIMMLER, H. SCHMECK, AND H. SCHRÖDER, *Systolic sorting on a mesh-connected network*, IEEE Trans. Comput., 14 (1985), pp. 652-658.

[14] H. LEV-ARI, *Modular computing networks: a new methodology for analysis and design of parallel algorithms/architectures*, Report #29, Integrated Systems Inc., Palo Alto, CA, December 1983.

[15] R. MELHEM AND W. C. RHEINBOLDT, *A mathematical model for the verification of systolic arrays*, SIAM J. Comput., 13 (1984), pp. 541-565.

[16] S. K. RAO, *Regular iterative algorithms and their implementations on processor arrays*, Ph.D. thesis, Department of Electrical Engineering, Stanford University, Stanford, CA, October 1985.
[17] E. TIDÉN, *Verification of systolic arrays—a case study*, Tech. Report TRITA-NA-8403, Department of Numerical Analysis and Computing Science, The Royal Institute of Technology, Sweden, 1984.
[18] J. D. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 644-658, August 1990

004

PARTIAL TYPES AND INTERVALS*

M. DEZANI-CIANCAGLINI†‡ AND B. VENNERI†

Abstract. The main idea of this paper is to develop an inference system to assign *partial types* to terms of the untyped lambda calculus. A term can either be *necessarily* or *possibly* of a certain type; these notions of necessity and possibility are incorporated into the type inference system as *modalities.* A subclass of types are the *total* types, for which necessity and possibility are equivalent. In the semantics the meaning of a total type is a set of values in the domain, the meaning of a type being, in general, an *interval* (a set of sets of values). This is a generalization of Cartwright's semantics [Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1984, pp. 22-36]. The main results are the *soundness* and *completeness* of the type inference system with respect to interval semantics.

Key words. type inference, partial types, modalities, type interpretation, intervals, soundness, completeness

AMS(MOS) subject classifications. 03B15, 03B40, 03B50

Introduction. One of the most interesting approaches for studying polymorphic type disciplines for functional programming is the predicate-type theory for λ-calculus, in which types are assigned to untyped terms by means of a system of inference rules.

A natural way of defining the semantics of a predicate-type theory is to interpret the underlying untyped language in a domain D and types as subsets of D, in such a way that the interpretation of a term is contained in the interpretation of every type that can be assigned to that term. The interpretation of types as arbitrary subsets of D presents problems when dealing with polymorphic types, since we can deduce types (built by the universal quantifier) whose interpretation in each model is the empty set. In this case the usual inference rules are no more sound (alternative inference rules that allow empty interpretations of types are discussed in [11] and [12]).

The solution proposed by MacQueen, Plotkin, and Sethi in [8] is to assume that D is a Scott domain and to interpret types as *ideals*, i.e., nonempty subsets of D closed under approximations and limits of increasing chains. It turns out however that even the theory of ideals is not entirely satisfactory, since the function type constructor is not monotonic with respect to the inclusion of ideals. This entails that, if the type system includes recursively defined types, there is no straightforward way of interpreting them as fixed points of *monotonic* mappings over the space of ideals, exploiting standard techniques available in such a context. Reference [8] establishes that unique solutions exist for an important syntactic class of recursive type equations (formally called contractive equations) by defining a metrics on ideals. This result is generalized in [4] by means of a different approach, according to which the interpretation of a type is obtained as a suitably defined limit of a denumerable sequence of approximate interpretations, whose construction closely parallels the structure of the underlying domain.

* Received by the editors December 23, 1987; accepted for publication (in revised form) May 23, 1989. This work was partially supported by Jumelage Lambda Calcul Type, CEE Project No. ST2J-0374-C and by M.P.I. 40% Comitato per la Matematica.

† Dipartimento di Informatica, Università di Torino, Corso Svizzera 185, 10149 Torino, Italy.

‡ This author was formerly affiliated with Centro Linceo Interdisciplinare di Scienze Matematiche e loro Applicazioni (for the academic years 1984-85, 1985-86, and 1986-87).

To overcome the limitations of previous models, in [5] Cartwright proposes a new interpretation of types, i.e., the interpretation of types as *intervals*, which subsumes the interpretation of types as ideals. The set of intervals contains the set of ideals as a proper subset while all type constructors turn out to be continuous functions. In particular, we can use the familiar Tarski least fixed-point construction for solving recursive type equations.

The main advantages of such a new approach versus the ideal model are discussed in [5]; the motivation behind Cartwright's results is essentially that all of the standard type operations on ideals have natural extensions to the space of intervals, which are computable.

However the key idea underlying the formalization of types as intervals seems appealing and fruitful by itself, so that it is natural to investigate the value of an interval predicate-type theory for functional programming languages. To this end, the first step is the formulation of a type inference system in which all features of types as intervals are incorporated in the most general way. This is the topic of the present paper. Note that Cartwright's semantics is not sound for standard type inference systems.

In this paper, a generalized notion of interval and type interpretation will be introduced; both Cartwright's semantics and intervals built on the term model turn out to be particular cases. Starting from any domain D of a λ-model, the intervals will be defined simply as subsets of 2^D satisfying some minimal conditions. In fact, there is a certain amount of freedom in choosing the collection of intervals on which the type interpretation is built.

The notion of types as sets of sets of values (intervals), instead of sets of values, leads to two different forms of membership; a value d belongs *possibly* or *necessarily* to an interval I according to whether d belongs to *some* or to *every* element in I. These different memberships will be represented in the formal system as *modalities*; this implies that types assigned to terms may be *partial*.

Intervals can be ordered by the inverse of the set inclusion relation reflecting their information content. Intuitively, an interval I is less informative than an interval J whenever I contains all the elements of J, and possibly more.

An interesting subclass of intervals are those that are *maximal* elements in this ordering (they contain just one set). The syntactic counterpart of a maximal interval is a *total* type; totality is characterized by the fact that necessity and possibility are equivalent. A partial type σ approximates many different total types, all types whose interpretations belong to the interval which is the meaning of σ. The statement "a term M possibly belongs to some partial type σ whose meaning is the interval I" means that the interpretation of M belongs to some set in I, and possibly to everyone.

All these features lead to a finer characterization of term behaviours than in standard-type systems. The total typing of a term coincides with that of standard systems, whereas partial types allow us to type terms whose functional properties are not completely known. Moreover, it can be desirable to pass from a total typing to an approximation of it in order to represent how the term *inherits* properties of a less informative type (this is briefly discussed in § 5).

This paper proposes an inference system to assign partial types to terms of the untyped λ-calculus and proves its main structural properties. The formal system is introduced in § 1. In § 2 the generalized interval semantics is presented and its relationship with Cartwright's semantics are discussed. The inference rules are proved *sound* and *complete* for this generalized semantics in § 3. As for completeness, an interpretation of types as intervals on the term model of β-equality is built. Section 4 discusses the relationship between the system presented in this paper and standard inference systems.

1. Type inference. The set Λ of *terms* is defined by

$$M := x|MN|\lambda x.M$$

where x ranges over the set of terms variables. We always consider terms modulo α-conversion. β-reduction $\twoheadrightarrow_\beta$, β-equality $=_\beta$, and η-reduction $\twoheadrightarrow_\eta$ between terms are defined as usual (cf. [1]).

The set $\mathscr{T}$ of *types* is defined by the grammar

$$\sigma := \varphi|\sigma \to \rho|\forall\varphi.\sigma|\mu\varphi.\sigma$$

where φ ranges over type variables. $Q\vec{\varphi}$ is short for $Q_1\varphi_1 \cdots Q_n\varphi_n$ where $Q_i \in \{\forall, \mu\}$. Types that differ only in the names of bound variables are identified.

We define an equivalence relation between types that takes into account the interpretation of μ as fixed-point operator.

DEFINITION 1.1. $\sim$ is the reflexive, symmetric and transitive closure of the smallest relation satisfying

$$\mu\varphi.\sigma \sim \sigma[\mu\varphi.\sigma/\varphi]$$

$$\sigma \sim \rho \Rightarrow \mu\varphi.\sigma \sim \mu\varphi.\rho, \quad \forall\varphi.\sigma \sim \forall\varphi.\rho$$

$$\sigma_1 \sim \sigma_2, \rho_1 \sim \rho_2 \Rightarrow \sigma_1 \to \rho_1 \sim \sigma_2 \to \rho_2.$$

We want to introduce a proper subset of $\mathscr{T}$, the subset of *total types*, whose meanings will be maximal intervals (i.e. intervals that contain only one set; see Definition 2.1(iv)). Actually, the results of the present paper hold for many definitions of total types. The only constraints are

(1) If τ is closed and does not contain subtypes of the shape $\mu\varphi.\sigma$, where σ contains at least one free occurrence of φ, then τ is total.

(2) If σ is open or σ contains subtypes of the shape $\mu\varphi Q\vec{\psi}.\varphi$ then σ is not total.

(3) If τ is total and $\tau \sim \tau'$ then τ' is total.

(4) If $\forall\varphi.\sigma$ and τ are total then $\sigma[\tau/\varphi]$ is total.

Condition (1) is justified by the observation that it is precisely the recursive types that make it necessary to deal with intervals containing more than one set. Condition (2) is required since we want total types corresponding to maximal intervals in Cartwright's semantics (the only closed types that are not total in [5] are those containing $\mu\varphi Q\vec{\psi}.\varphi$). The preservation of totality under $\sim$ and under substitution of quantified variables by total types (conditions (3) and (4)) agrees with rules ($\sim$) and ($\forall E!$) of the inference system (Definition 1.3).

Therefore, in the following we will assume an arbitrary fixed definition of total types that satisfies the above conditions. The *modalities* are !, ?. Let ξ range over {!, ?}.

DEFINITION 1.2. (i) A *modal typing assertion* is an expression $M : \xi\sigma$ where $\xi \in$ {!, ?} is the *modality*, $M \in \Lambda$ is the *subject*, and $\sigma \in \mathscr{T}$ is the *predicate*.

(ii) A *basis* B is a set of modal typing assertions such that all subjects are distinct term variables.

(iii) A *modal typing statement* is an expression $B \vdash M : \xi\sigma$ that can be derived by the inference axioms and rules of Definition 1.3.

$B \vdash M : \xi\sigma$ may be read as "the term M has *possibly* (if $\xi = ?$) or *necessarily* (if $\xi = !$) type σ with respect to B." We define $\bar{\xi} = !$ if $\xi = ?$, ? otherwise.

We now introduce the system of inference rules that are formulated in a natural deduction style. A type variable φ is *bindable* in M with respect to B if φ does not occur free in any assertion of B whose subject is a free variable of M.

DEFINITION 1.3 (Type inference system). The axioms and rules to derive modal typing statements are

(Ax) $\quad B \vdash x : \xi\sigma$ if $x : \xi\sigma \in B$,

$(!\Rightarrow ?)$ $\quad \dfrac{B \vdash M : !\sigma}{B \vdash M : ?\sigma}$,

$(?\Rightarrow !)$ $\quad \dfrac{B \vdash M : ?\tau}{B \vdash M : !\tau}$ if τ is total,

$(\rightarrow I)$ $\quad \dfrac{B \cup \{x : \bar{\xi}\sigma\} \vdash M : \xi\rho}{B \vdash \lambda x.M : \xi\sigma \rightarrow \rho}$ if x does not occur in B,

$(\rightarrow E)$ $\quad \dfrac{B \vdash M : \xi\sigma \rightarrow \rho \qquad B \vdash N : \bar{\xi}\sigma}{B \vdash MN : \xi\rho}$,

$(\forall I)$ $\quad \dfrac{B \vdash M : \xi\sigma}{B \vdash M : \xi\forall\varphi.\sigma}$ if φ is bindable in M with respect to B,

$(\forall E ?)$ $\quad \dfrac{B \vdash M : \xi\forall\varphi.\sigma}{B \vdash M : ?\sigma[\rho/\varphi]}$,

$(\forall E !)$ $\quad \dfrac{B \vdash M : !\forall\varphi.\sigma}{B \vdash M : !\sigma[\tau/\varphi]}$ if τ is total,

$(\sim)$ $\quad \dfrac{B \vdash M : \xi\sigma}{B \vdash M : \xi\rho}$ if $\sigma \sim \rho$,

(μI) $\quad \dfrac{B \vdash M : \xi\sigma[\mu\varphi.\sigma/\varphi]}{B \vdash M : \xi\mu\varphi.\sigma}$,

(μE) $\quad \dfrac{B \vdash M : \xi\mu\varphi.\sigma}{B \vdash M : \xi\sigma[\mu\varphi.\sigma/\varphi]}$.

Rules (μI) and (μE) are superfluous, since they can be directly derived from $(\sim)$. Instead, (μI) and (μE) do not imply $(\sim)$ also for types in which $\forall$ does not occur, since for example we cannot derive $\{x : \xi\varphi \rightarrow \mu\psi.\sigma\} \vdash x : \xi\varphi \rightarrow \sigma[\mu\psi.\sigma/\psi]$ without rule $(\sim)$. Rule $(!\Rightarrow ?)$ formalizes the fact that each element belonging to all sets of an interval belongs, of course, to some of them. Rule $(?\Rightarrow !)$ means that necessity and possibility are equivalent for total types; it is sound since total types correspond to maximal intervals in the semantics.

It is interesting to observe that rules $(\rightarrow I)$ and $(\rightarrow E)$ correspond to the rules of introduction and elimination of implication of one of the systems introduced by Girard in [6] (more precisely, the three-valued semantics in the deterministic case). Note the role played by modalities in our rules. For example, if the $\bar{\xi}$ occurring in $(\rightarrow I)$ and $(\rightarrow E)$ were changed to a ξ then we could derive $\vdash \lambda x.x : !\rho \rightarrow \rho$ for all types ρ, which is not sound for interval semantics (see Remark 2 in § 3).

Rule $(\forall E ?)$ can yield only statements whose modality is "?," but this does not happen in three-valued models (cf. [6]). Our choice has been motivated by the soundness requirement with respect to the interpretation of types as intervals (see Remark 2). However, if the quantified variable is substituted by a total type, the modality in the conclusion is not weakened, according to rule $(\forall E !)$.

Rules (μI) and (μE) are standard and $(\sim)$ takes into account the equivalence relation between polymorphic recursive types.

Note that if $B \vdash M : \xi\sigma$ then also $B' \vdash M : \xi\sigma$ for all $B' \supseteq B$. Moreover, $B \vdash M : \xi\sigma$ implies $B \restriction M \vdash M : \xi\sigma$ where $B \restriction M = \{x : \xi'\rho \mid x : \xi'\rho \in B \text{ and } x \text{ occurs free in } M\}$. These properties will be widely used in the following proofs.

We now prove the subject reduction theorem. Subject expansion instead does not hold; for example,

$$\{x : !\varphi\} \vdash x : !\varphi \quad \text{but} \quad \{x : !\varphi\} \nvdash (\lambda y.y)x : !\varphi.$$

To show how types and modalities are preserved by β-reduction, we need some technical devices. First we must introduce the notions of *instance* $(\leqq)$ and *total instance* $(\leqq_T)$ of a type. Informally, an instance is total if and only if all variables bound by $\forall$ are replaced by total types. Then we prove, in Lemma 1.5, some structural properties of deductions, taking into account the given notions of instance.

DEFINITION 1.4 (Instance). (i) $\leqq$ is the reflexive and transitive closure of the smallest relation satisfying

— $\forall\varphi.\sigma \leqq \sigma[\rho/\varphi]$ for all types ρ,

— $\sigma \sim \sigma'$, $\rho \sim \rho'$ and $\sigma' \leqq \rho' \Rightarrow \sigma \leqq \rho$.

(ii) $\leqq_T$ is the reflexive and transitive closure of the smallest relation satisfying

— $\forall\varphi.\sigma \leqq_T \sigma[\tau/\varphi]$ for all total types τ,

— $\sigma \sim \sigma'$, $\rho \sim \rho'$, and $\sigma' \leqq_T \rho' \Rightarrow \sigma \leqq_T \rho$.

From the second clause of the definition of $\leqq$ $(\leqq_T)$ we have immediately that $\sigma \sim \rho$ implies $\sigma \leqq \rho$ $(\sigma \leqq_T \rho)$.

Let $B[\rho/\varphi] = \{x : \xi\sigma[\rho/\varphi] \mid x : \xi\sigma \in B\}$ and $B/x = \{y : \xi\sigma \mid y : \xi\sigma \in B \text{ and } y \not\equiv x\}$.

LEMMA 1.5. (i) $B \vdash M : \xi\sigma \Rightarrow B[\rho/\varphi] \vdash M : \xi\sigma[\rho/\varphi]$.

(ii) $B \vdash \lambda x.M : \xi\sigma \Rightarrow \sigma \equiv Q\vec{\varphi}.\rho \to \nu$ *for suitable* $\vec{\varphi}$, ρ, *and* ν.

(iii) $B \vdash \lambda x.M : \xi\sigma$ *and* $\sigma \leqq \rho \to \nu \Rightarrow B/x \cup \{x : !\rho\} \vdash M : ?\nu$.

(iv) $B \vdash \lambda x.M : !\sigma$ *and* $\sigma \leqq_T \rho \to \nu \Rightarrow B/x \cup \{x : ?\rho\} \vdash M : !\nu$.

Proof. Part (i) is easily proved by induction on derivations. For rule $(\sim)$ note that $\sigma \sim \nu$ implies $\sigma[\rho/\varphi] \sim \nu[\rho/\varphi]$.

Part (ii) is straightforward by induction on derivations.

Part (iii) is proved by induction on derivations. If the last applied rule is $(\forall I)$ we have

$$(\forall I) \qquad \frac{B \vdash \lambda x.M : \xi\sigma'}{B \vdash \lambda x.M : \xi\forall\varphi.\sigma'} \quad \text{where } \varphi \text{ is bindable in } \lambda x.M \text{ with respect to } B.$$

It is easy to verify that if $\forall\varphi.\sigma' \leqq \rho \to \nu$, where by (ii) $\sigma' \equiv Q\vec{\psi}.\sigma_1 \to \sigma_2$, then $\rho \equiv \rho'[\zeta/\varphi]$, $\nu \equiv \nu'[\zeta/\varphi]$ and $\sigma' \leqq \rho' \to \nu'$ for suitable ρ', ν', ζ. So by the induction hypothesis $B/x \cup \{x : !\rho'\} \vdash M : ?\nu'$ and by (i) $B/x \cup \{x : !\rho\} \vdash M : ?\nu$ since φ is bindable in $\lambda x.M$ with respect to B. For rules $(\forall E?)$ and $(\forall E!)$ note that $\sigma'[\zeta/\varphi] \leqq \rho \to \nu$ implies that $\forall\varphi.\sigma' \leqq \rho \to \nu$. The other cases are trivial.

Part (iv). Again by induction on derivations. If the last applied rule is $(?\Rightarrow!)$ we have $\sigma \equiv \tau$ where τ is a total type and

$$(?\Rightarrow!) \qquad \frac{B \vdash \lambda x.M : ?\tau}{B \vdash \lambda x.M : !\tau}.$$

$\tau \leqq_T \rho \to \nu$ implies $\tau \leqq \rho \to \nu$ and therefore by (ii) $B/x \cup \{x : !\rho\} \vdash M : ?\nu$. $\tau \leqq_T \rho \to \nu$ and τ total imply ρ, ν total by conditions (3) and (4) on the definition of total types. Therefore we obtain a derivation of $B/x \cup \{x : ?\rho\} \vdash M : !\nu$ by replacing, in a derivation

of $B/x \cup \{x:!\rho\} \vdash M:?\nu$, $B/x \cup \{x:!\rho\} \vdash x:!\rho$ by

$$(?\Rightarrow!) \qquad \frac{B/x \cup \{x:?\rho\} \vdash x:?\rho}{B/x \cup \{x:?\rho\} \vdash x:!\rho}$$

and by applying rule $(?\Rightarrow!)$ to the conclusion.

If the last applied rule is $(\forall I)$ then the proof runs as in case (ii). If the last applied rule is $(\forall E!)$ note that $\sigma'[\tau/\varphi] \leqq_T \rho \to \nu$ and τ total imply $\forall\varphi.\sigma' \leqq_T \rho \to \nu$. The other cases are trivial. □

THEOREM 1.6 (Subject reduction). *If $B \vdash M:\xi\sigma$ and $M \twoheadrightarrow_\beta N$, then $B \vdash N:\xi\sigma$.*

Proof. Clearly, it is sufficient to prove that $B \vdash (\lambda x.M)N:\xi\sigma$ implies $B \vdash M[N/x]:\xi\sigma$. This proof can be done by induction on derivations. The only interesting case is rule $(\to E)$. If the last step is

$$(\to E) \qquad \frac{B \vdash \lambda x.M:\xi\rho \to \sigma \qquad B \vdash N:\bar{\xi}\rho}{B \vdash (\lambda x.M)N:\xi\sigma},$$

then $B \vdash \lambda x.M:\xi\rho \to \sigma$ implies $B/x \cup \{x:\bar{\xi}\rho\} \vdash M:\xi\sigma$ by Lemma 1.5(iii) and (iv). Therefore we can obtain a derivation of $B \vdash M[N/x]:\xi\sigma$ simply by replacing $B/x \cup \{x:\bar{\xi}\rho\} \vdash x:\bar{\xi}\rho$ by a derivation of $B \vdash N:\bar{\xi}\rho$ and x by N in a derivation of $B/x \cup \{x:\bar{\xi}\rho\} \vdash M:\xi\sigma$. □

2. Semantics. We interpret terms and types independently of each other in the domain of a λ-model. The semantics of terms is standard, whereas the meanings of types are sets of nonempty sets of values.

Let us recall (cf. [10] and [1]) that a λ-model $\mathcal{M} = \langle D, \cdot, \boldsymbol{\varepsilon}\rangle$ is a set D together with a binary operation $\cdot$ and elements $\mathbf{K}, \mathbf{S} \in D$ such that

$$(\mathbf{K} \cdot d) \cdot e = d,$$

$$((\mathbf{S} \cdot d) \cdot e) \cdot f = (d \cdot f) \cdot (e \cdot f),$$

$$(\boldsymbol{\varepsilon} \cdot d) \cdot e = d \cdot e,$$

$$\forall e(d_1 \cdot e = d_2 \cdot e) \Rightarrow \boldsymbol{\varepsilon} \cdot d_1 = \boldsymbol{\varepsilon} \cdot d_2$$

$$\boldsymbol{\varepsilon} \cdot \boldsymbol{\varepsilon} = \boldsymbol{\varepsilon}.$$

Given a λ-model $\mathcal{M} = \langle D, \cdot, \boldsymbol{\varepsilon}\rangle$ and a mapping (*environment*) θ: Term Variables $\to D$, the meaning of a term in $\mathcal{M}$ is inductively defined by

$$[\![x]\!]^{\mathcal{M}}_\theta = \theta(x),$$

$$[\![MN]\!]^{\mathcal{M}}_\theta = [\![M]\!]^{\mathcal{M}}_\theta \cdot [\![N]\!]^{\mathcal{M}}_\theta,$$

$$[\![\lambda x.M]\!]^{\mathcal{M}}_\theta = \boldsymbol{\varepsilon} \cdot d \quad \text{where } d \cdot e = [\![M]\!]^{\mathcal{M}}_{\theta[e/x]} \text{ for all } e \in D.$$

To interpret types we introduce the notion of intervals, i.e., sets of sets of values, and some useful operations on them. The operations on intervals are defined using the $\cap$ and $\to$ operations on subsets of D. $\cap$ is set theoretic intersection and $\to$ (the function space constructor) must satisfy the condition (cf. [12])

$$\{d \in D \mid d \cdot A \subseteq B\} \cap \mathcal{F} \subseteq A \to B \subseteq \{d \in D \mid d \cdot A \subseteq B\}$$

where $d \cdot A$ is short-hand for the set $\{d \cdot e \mid e \in A\}$ and $\mathcal{F}$ is the range of $\boldsymbol{\varepsilon}$. In the particular cases

$$A \to B = \{d \in D \mid d \cdot A \subseteq B\} \quad \text{and} \quad A \to B = \{d \in D \mid d \cdot A \subseteq B\} \cap \mathcal{F}$$

we have the constructors $\to_S$ and $\to_{\mathscr{F}}$, respectively. To build a domain of intervals we must choose a subset $\mathscr{S} \subseteq 2^D$ and a relation $\mathscr{R} \subseteq \mathscr{S} \times \mathscr{S}$, which is a subset of the inclusion relation (i.e., such that $\langle A, B\rangle \in \mathscr{R}$ implies $A \subseteq B$). An obvious choice is $\mathscr{S} = 2^D$ and $\mathscr{R} = \subseteq$.

DEFINITION 2.1 (Intervals and their operations). Let $\mathscr{M} = \langle D, \cdot, \mathbf{\varepsilon}\rangle$ be a λ-model, $\mathscr{S} \subseteq 2^D$ and $\mathscr{R} \subseteq \mathscr{S} \times \mathscr{S}$ be such that $\langle A, B\rangle \in \mathscr{R}$ implies $A \subseteq B$.

(i) $\langle A, B\rangle \in \mathscr{R}$ defines the interval $|A, B|$ on $\mathscr{S}$ and $\mathscr{R}$ as

$$|A, B| = \{C \in \mathscr{S} \mid A \subseteq C \subseteq B\}.$$

(ii) Let $Int(\mathscr{S}, \mathscr{R})$ be the set of intervals on $\mathscr{S}$ and $\mathscr{R}$.

(iii) The partial functions $\sqcap$ and $\Rightarrow: Int(\mathscr{S}, \mathscr{R}) \times Int(\mathscr{S}, \mathscr{R}) \to Int(\mathscr{S}, \mathscr{R})$ are defined by

$$|A, B| \sqcap |C, E| = |A \cap C, B \cap E| \quad \text{if } \langle A \cap C, B \cap E\rangle \in \mathscr{R},$$

$$|A, B| \Rightarrow |C, E| = |B \to C, A \to E| \quad \text{if } \langle B \to C, A \to E\rangle \in \mathscr{R}.$$

(iv) An interval $|A, B|$ is *maximal* if and only if $A = B$.

Note that $A \subseteq B$ and $C \subseteq E$ imply $B \to C \subseteq A \to E$. Using $\to_S$, $\to_{\mathscr{F}}$ instead of $\to$ in Definition 2.1(iii) we obtain the definitions of $\Rightarrow_S$ and $\Rightarrow_{\mathscr{F}}$, respectively.

The order relation on intervals is the superset relation on sets, i.e.,

$$|A, B| \sqsubseteq |C, E| \quad \text{iff } A \subseteq C \text{ and } E \subseteq B.$$

$Int(2^D, \subseteq)$ is a c.p.o. with respect to $\sqsubseteq$ whose bottom is $|\varnothing, D|$ and whose maximal elements are the maximal intervals. Note that $\Rightarrow$ is monotonic in both arguments. If I is an element of $Int(\mathscr{S}, \mathscr{R})$, say $I = |A, B|$, then $I^! = A$ and $I^? = B$.

An interpretation of types as intervals will be a mapping from types to $Int(\mathscr{S}, \mathscr{R})$ satisfying suitable conditions, expressed by the following definition. Note that not all choices of $\mathscr{S}$ and $\mathscr{R}$ allow the construction of type interpretations.

DEFINITION 2.2 (Type interpretation). A mapping $[\![\ ]\!]: \mathscr{T} \to \mathrm{Env} \to Int(\mathscr{S}, \mathscr{R})$, where $\mathrm{Env} = \text{Type Variables} \to Int(\mathscr{S}, \mathscr{R})$ is a *type interpretation* if and only if for all $\eta \in \mathrm{Env}$

(1) $[\![\varphi]\!]_\eta = \eta(\varphi)$,
(2) $[\![\sigma]\!]_\eta$ does not contain the empty set,
(3) τ total $\Rightarrow [\![\tau]\!]_\eta$ maximal,
(4) $\sigma \sim \rho \Rightarrow [\![\sigma]\!]_\eta = [\![\rho]\!]_\eta$,
(5) $[\![\sigma \to \rho]\!]_\eta = [\![\sigma]\!]_\eta \Rightarrow [\![\rho]\!]_\eta$,
(6) $[\![\forall\varphi.\sigma]\!]_\eta^? \subseteq \bigcap_{\rho \in \mathscr{T}} [\![\sigma[\rho/\varphi]]\!]_\eta^?$,
(7) $[\![\forall\varphi.\sigma]\!]_\eta^! \subseteq \bigcap_{\tau \text{ total}} [\![\sigma[\tau/\varphi]]\!]_\eta^!$,
(8) $[\![\forall\varphi.\sigma]\!]_\eta^\xi \supseteq \bigcap_{I \in Int(\mathscr{S}, \mathscr{R})} ([\![\sigma]\!]_{\eta[I/\varphi]})^\xi$.

We say that $[\![\ ]\!]$ is a *simple type interpretation* (or, respectively, an *$\mathscr{F}$-type interpretation*) if and only if $[\![\sigma \to \tau]\!]_\eta = [\![\sigma]\!]_\eta \Rightarrow_S [\![\tau]\!]_\eta$ ($[\![\sigma \to \tau]\!]_\eta = [\![\sigma]\!]_\eta \Rightarrow_{\mathscr{F}} [\![\tau]\!]_\eta$). A type interpretation is *total* if and only if $[\![\forall\varphi.\sigma]\!]_\eta = \bigsqcap_{\tau \text{ total}} [\![\sigma[\tau/\varphi]]\!]_\eta$. In this case by condition (6) we have $\bigcap_{\rho \in \mathscr{T}} [\![\sigma[\rho/\varphi]]\!]_\eta^? = \bigcap_{\tau \text{ total}} [\![\sigma[\tau/\varphi]]\!]_\eta^?$.

Conditions (1) and (5) are standard. Condition (2) will be justified in Remark 1, § 3. Conditions (3) and (4) take into account our requirements about totality and equivalence of types, respectively. As an immediate consequence of condition (4) we have the usual interpretation of μ as the fixed-point operator. In fact, $[\![\mu\varphi.\sigma]\!]_\eta = [\![\sigma[\mu\varphi.\sigma/\varphi]]\!]_\eta$ since $\mu\varphi.\sigma \sim \sigma[\mu\varphi.\sigma/\varphi]$. Note that the requirement that the meanings of total types are maximal intervals prevents (in some cases) μ to choose the least solution.

Conditions (6), (7), and (8) give us some freedom in the choice of the range of $\forall$. We want to allow different interpretations in which this range can be the set of values of total types, or the set of maximal intervals (as in Cartwright's semantics), or the set of all intervals. In fact, these conditions imply

$$\bigcap_{I \in Int(\mathscr{S},\mathscr{R})} (\llbracket \sigma \rrbracket_{\eta[I/\varphi]})^? \subseteq \llbracket \forall \varphi.\sigma \rrbracket_\eta^? \subseteq \bigcap_{\rho \in \mathscr{T}} \llbracket \sigma[\rho/\varphi] \rrbracket_\eta^?$$

and

$$\bigcap_{I \in Int(\mathscr{S},\mathscr{R})} (\llbracket \sigma \rrbracket_{\eta[I/\varphi]})^! \subseteq \llbracket \forall \varphi.\sigma \rrbracket_\eta^! \subseteq \bigcap_{\tau \text{ total}} \llbracket \sigma[\tau/\varphi] \rrbracket_\eta^!.$$

Following the method of [12] we can construct a number of sets $\mathscr{S}$ closed under $\cap$, $\to$ that do not contain the empty set $\varnothing$. Given a λ-model $\mathscr{M} = \langle D, \cdot, \varepsilon \rangle$ we say that a nonempty subset $Z \subseteq D$ is a zero set if and only if $\forall z \in Z\ \forall d \in D,\ z \cdot d \in Z$. It is easy to verify that for any zero set Z the set $\mathscr{S} = \{A \subseteq D \mid Z \subseteq A\}$ is closed under $\cap$, $\to$ and does not contain $\varnothing$. In [12] a list of λ-models and zero sets is given.

We discuss now the meaning of types given by Cartwright and we prove that it is a type interpretation according to Definition 2.2. The language considered in [5] is the untyped λ-calculus equipped with a suitable set of constants. Therefore the semantic domain D^* consists of a disjoint collection of subspaces, such as truth values, integers, tuples, and functions. We refer to [5] for the details of the construction of D^*; let us only recall that D^* is a consistently complete, ω-algebraic c.p.o. Obviously D^* can be turned into a λ-model simply by defining $d \cdot e = \bot$ whenever d is not a function. Let $\mathscr{S}$ be the set Ide_{D^*} of the ideals (i.e., nonempty closed sets with respect to Scott topology) on D^* and $\mathscr{R}$ be $\subseteq$. The function space constructor is $\Rightarrow_s$ (actually $\Rightarrow_s$ and $\Rightarrow_{\mathscr{F}}$ coincide for D^*). μ is interpreted as the least fixed-point operator and the parameter of $\forall$ ranges over maximal intervals. Actually, Cartwright also considers other constructors, such as the $\exists$ quantifier, which we could include without problems.

If $Type(D^*) = Int(\mathrm{Ide}_{D^*}, \subseteq)$, Cartwright's semantics is a mapping $\langle\!\langle\ \rangle\!\rangle : \mathscr{T} \to \mathrm{Env} \to Type(D^*)$ such that, for every environment $\eta \in \mathrm{Env} = Type\ \mathrm{Variables} \to Type(D^*)$

(i) $\langle\!\langle \sigma \to \rho \rangle\!\rangle_\eta = \langle\!\langle \sigma \rangle\!\rangle_\eta \Rightarrow_s \langle\!\langle \rho \rangle\!\rangle_\eta$.

(ii) $\langle\!\langle \mu\varphi.\sigma \rangle\!\rangle_\eta = \mathbf{Y}f$ where $\mathbf{Y} : [Type(D^*) \to Type(D^*)] \to Type(D^*)$ is the least fixed-point operator and $f(I) = \langle\!\langle \sigma \rangle\!\rangle_{\eta[I/\varphi]}$ for all $I \in Type(D^*)$.

(iii) $\langle\!\langle \forall \varphi.\sigma \rangle\!\rangle_\eta = \bigsqcap_{I \in \mathrm{Max}} \langle\!\langle \sigma \rangle\!\rangle_{\eta[I/\varphi]}$ where $\mathrm{Max} \subseteq Type(D^*)$ is the set of the maximal elements of $Type(D^*)$.

It is obvious that $\langle\!\langle\ \rangle\!\rangle$ satisfies conditions (1), (2), (4), and (5) of Definition 2.2. Condition (3) is actually proved in [5], since the present notion of total type implies the formal contractiveness as defined in [8]. In fact, in [5] Cartwright develops a metric space of intervals (based on the corresponding theory for ideals) to assure that the least solution to every formally contractive system of equations is a maximal interval. Note that this proof essentially depends on the presence of constants in the language. Condition (7) follows from condition (3). Condition (8) is verified since $\mathrm{Max} \subseteq Type(D^*)$.

We prove that condition (6) is satisfied in Theorem 2.3.

THEOREM 2.3. *Cartwright's semantics* $\langle\!\langle\ \rangle\!\rangle$ *is a type interpretation.*

Proof. We only need to prove condition (6) of Definition 2.2, i.e.,

$$\langle\!\langle \forall \varphi.\sigma \rangle\!\rangle_\eta^? \subseteq \bigcap_{\rho \in \mathscr{T}} \langle\!\langle \sigma[\rho/\varphi] \rangle\!\rangle_\eta^?.$$

To this end it is sufficient to observe that, for any $\rho \in \mathscr{T}$,

$$\langle\!\langle \sigma[\rho/\varphi] \rangle\!\rangle_\eta^? \supseteq \bigcap_{I \in \mathrm{Max}} \langle\!\langle \sigma \rangle\!\rangle_{\eta[I/\varphi]}^?.$$

In fact, by monotonicity, if $\langle\!\langle \rho \rangle\!\rangle_\eta = I$ and $J = |I^!, I^!|$ then $\langle\!\langle \sigma[\rho/\varphi] \rangle\!\rangle_\eta \sqsubseteq \langle\!\langle \sigma \rangle\!\rangle_{\eta[J/\varphi]}$ and therefore $\langle\!\langle \sigma[\rho/\varphi] \rangle\!\rangle^?_\eta \supseteq \langle\!\langle \sigma \rangle\!\rangle^?_{\eta[J/\varphi]}$. □

3. Soundness and completeness. The notion of semantic satisfiability is defined as usual, taking into account the two different forms of membership.

DEFINITION 3.1 (Semantic satisfiability). (i) $\mathcal{M}, \theta, [\![\ ]\!], \eta \models M : \xi\sigma$ if and only if $[\![M]\!]^{\mathcal{M}}_\theta \in [\![\sigma]\!]^\xi_\eta$.

(ii) $\mathcal{M}, \theta, [\![\ ]\!], \eta \models B$ if and only if $\forall x : \xi\sigma \in B$ $\mathcal{M}, \theta, [\![\ ]\!], \eta \models x : \xi\sigma$.

(iii) $B \models M : \xi\sigma$ if and only if $\forall \mathcal{M}, \theta, [\![\ ]\!], \eta$

$$[\mathcal{M}, \theta, [\![\ ]\!], \eta \models B \Rightarrow \mathcal{M}, \theta, [\![\ ]\!], \eta \models M : \xi\sigma],$$

$B \models_s M : \xi\sigma$ if and only if $\forall \mathcal{M}, \theta$, simple type interpretation $[\![\ ]\!], \eta$

$$[\mathcal{M}, \theta, [\![\ ]\!], \eta \models B \Rightarrow \mathcal{M}, \theta, [\![\ ]\!], \eta \models M : \xi\sigma],$$

$B \models_{\mathscr{F}} M : \xi\sigma$ if and only if $\forall \mathcal{M}, \theta, \mathscr{F}$-type interpretation $[\![\ ]\!], \eta$

$$[\mathcal{M}, \theta, [\![\ ]\!], \eta \models B \Rightarrow \mathcal{M}, \theta, [\![\ ]\!], \eta \models M : \xi\sigma],$$

$B \models_T M : \xi\sigma$ if and only if $\forall \mathcal{M}, \theta$, total type interpretation $[\![\ ]\!], \eta$

$$[\mathcal{M}, \theta, [\![\ ]\!], \eta \models B \Rightarrow \mathcal{M}, \theta, [\![\ ]\!], \eta \models M : \xi\sigma].$$

Remark 1. Condition (2) of Definition 2.2 is justified by soundness and completeness requirements. For example, if we allow $[\![\sigma]\!]_\eta = |\varnothing, A|$ for all η, some σ and A, then it turns out that $x : !\sigma \models M : \xi\rho$ is vacuously valid. In fact, no model and environment can satisfy $x : !\sigma$ since $[\![\sigma]\!]^!_\eta$ is always empty. Moreover, the usual property "$B \vdash M : \xi\sigma \Rightarrow B \restriction M \vdash M : \xi\sigma$" is not sound. Empty interpretations of types produce similar pathologies in standard type assignment systems; they can be allowed by modifying the inference system [12].

The soundness of the inference rules easily follows from the definition of type interpretation.

THEOREM 3.2 (Soundness). $B \vdash M : \xi\sigma \Rightarrow B \models M : \xi\sigma$.

Proof. This proof is by induction on derivations. $(!\Rightarrow ?)$ is proved by definition of interval. $(?\Rightarrow !)$, $(\sim)$, $(\forall I)$, $(\forall E?)$, and $(\forall E!)$ follow from Definition 2.2, namely, from points (3), (4), (8), (6), and (7), respectively. The remaining cases, $(\to I)$ and $(\to E)$ are straightforward by using Definition 2.2(5) and the definition of $\Rightarrow$ on intervals. □

Remark 2. We are now able to discuss the restriction on rule $(\forall E!)$. In fact, a rule of the shape

$$\frac{B \vdash M : !\forall\varphi.\sigma}{B \vdash M : !\sigma[\rho/\varphi]}$$

is not sound. To show this, it is sufficient to observe that in the particular case $M \equiv \lambda x.x$ and $\sigma \equiv \varphi \to \varphi$ we could derive $\vdash \lambda x.x : !\rho \to \rho$ for all types ρ. But $[\![\lambda x.x]\!]^{\mathcal{M}}_\theta \notin [\![\rho \to \rho]\!]^!_\eta = B \to A$ in the general case $[\![\rho]\!]_\eta = |A, B|$ whenever A is a proper subset of B.

To obtain a complete system we must add rule (Eq) of subject equality

$$\text{(Eq)} \qquad \frac{B \vdash M : \xi\sigma \qquad M =_\beta N}{B \vdash N : \xi\sigma}.$$

The soundness of this rule is straightforward. It is easy to verify that an equality postponement theorem holds, i.e., if we deduce $B \vdash M : \xi\sigma$ using rule (Eq) then there is $N =_\beta M$ such that we can deduce $B \vdash N : \xi\sigma$ without rule (Eq). This means that rule (Eq) does not increase the expressiveness of the inference system.

In the rest of the section $B \vdash M : \xi\sigma$ will denote a modal typing statement that can be derived in this new system.

We now build the set of intervals on the term model $\mathcal{M}(\beta)$ and we define a mapping from types to these intervals. Lemma 3.5 proves that this mapping is a type interpretation since it satisfies the conditions of Definition 2.2. This construction is used to state the completeness of the type inference system. Recall that the term model of β-equality $\mathcal{M}(\beta) = \langle \Lambda(\beta), \cdot, [\lambda xy.xy] \rangle$ is defined by

$$\Lambda(\beta) = \{[M] \mid M \text{ is a term}\} \quad \text{where } [M] = \{N \mid N =_\beta M\},$$

$$[M] \cdot [N] = [MN], \quad \text{and}$$

$$[\![M]\!]_\theta^{\mathcal{M}(\beta)} = [M[M_1/x_1, \cdots, M_n/x_n]]$$

where $\theta(x_j) = [M_j]$ and $x_1, \cdots, x_n$ are the free variables of M. As proved in [7], given a term M and a basis B, we can assume that there is a basis $B^+ \supseteq B$ that contains infinitely many statements $x_{\sigma,\xi,i} : \xi\sigma$ for all $\sigma \in \mathscr{T}$, $\xi \in \{!, ?\}$, and $i \in \omega$, where the variables $x_{\sigma,\xi,i}$ are all distinct and do not occur in B and M. We refer to [7] for the technical details. It is easy to verify that $B \vdash M : \xi\sigma \Leftrightarrow B^+ \vdash M : \xi\sigma$.

The following definitions of intervals and type interpretation are done for a fixed basis B.

DEFINITION 3.3 (Intervals on the term model).

(i) $S(\xi, \sigma) = \{[M] \mid B^+ \vdash M : \xi\sigma\}$.

(ii) $R_{\Lambda(\beta)} = \{\langle S(!, \sigma), S(?, \sigma)\rangle \mid \sigma \in \mathscr{T}\}$.

(iii) $Type(\Lambda(\beta)) = Int(2^{\Lambda(\beta)}, R_{\Lambda(\beta)})$.

The inference system naturally induces an equivalence relation ($\approx$) between the types that can be assigned to the same set of terms, that is

$$\sigma \approx \rho \Leftrightarrow \forall B, M, \xi : B \vdash M : \xi\sigma \Leftrightarrow B \vdash M : \xi\rho.$$

For example, $\xi\varphi \approx \xi\forall\psi.\varphi$. Obviously $\approx$ contains $\sim$. Let $[\sigma]$ denote the equivalence class of σ under $\approx$, i.e., $[\sigma] = \{\rho \mid \rho \approx \sigma\}$. We will write $B \vdash M : \xi[\sigma]$ as short for $B \vdash M : \xi\rho$ for some $\rho \in [\sigma]$ (this implies $B \vdash M : \xi\rho$ for all $\rho \in [\sigma]$). Let $I(\sigma) = |S(!, \sigma), S(?, \sigma)|$. Clearly, $\sigma \approx \rho$ if and only if $I(\sigma) = I(\rho)$. Therefore an environment η : Type Variables $\to Type(\Lambda(\beta))$ associates a type variable with a type modulo $\approx$. We use this fact to define a type interpretation on $Type(\Lambda(\beta))$.

DEFINITION 3.4. (i) $\bar{\eta}(\sigma) = [\sigma[\sigma_1/\varphi_1, \cdots, \sigma_n/\varphi_n]]$ where $\eta(\varphi_j) = I(\sigma_j)$ and $\varphi_1, \cdots, \varphi_n$ are the free variables of σ.

(ii) $[\![\ ]\!] : \mathscr{T} \to \text{Env} \to Type(\Lambda(\beta))$ is defined by $[\![\sigma]\!]_\eta = I(\rho)$ for some $\rho \in \bar{\eta}(\sigma)$.

(iii) η_0 : Type Variables $\to Type(\Lambda(\beta))$ is defined by $\eta_0(\varphi) = I(\varphi)$.

(iv) θ_0 : Term Variables $\to \Lambda(\beta)$ is defined by $\theta_0(x) = [x]$.

Definition 3.4(ii) is sound since $I(\rho) = I(\rho')$ for all $\rho, \rho' \in \bar{\eta}(\sigma)$. It is easy to verify that $\bar{\eta}_0(\sigma) = [\sigma]$ (which implies that $[\![\sigma]\!]_{\eta_0} = I(\sigma)$) and $[\![M]\!]_{\theta_0}^{\mathcal{M}(\beta)} = [M]$.

Note that $[M] \in [\![\sigma]\!]_\eta^\xi$ if and only if $B^+ \vdash M : \xi\bar{\eta}(\sigma)$.

LEMMA 3.5. (i) $[\![\ ]\!]$ is a type interpretation.

(ii) $\mathcal{M}(\beta), \theta_0, [\![\ ]\!], \eta_0 \models B$.

Proof. (i) We prove that $[\![\ ]\!]$ satisfies conditions (1)-(8) of Definition 2.2.

Condition (1). $\eta(\varphi) = I(\sigma)$ implies $\bar{\eta}(\varphi) = [\sigma]$. By definition $[\![\varphi]\!]_\eta = I(\rho)$ for some $\rho \in \bar{\eta}(\varphi)$ and so we conclude $[\![\varphi]\!]_\eta = \eta(\varphi)$.

Condition (2). $[x_{\bar{\eta}(\sigma),\xi,i}] \in [\![\sigma]\!]_\eta^\xi$ for all i.

Condition (3). Proved by rule ($?\Rightarrow !$).

Condition (4). Proved by rule ($\sim$).

Condition (5). Recall that $[\![\sigma \to \rho]\!]_\eta = |[\![\sigma]\!]_\eta^? \to [\![\rho]\!]_\eta^!,\ [\![\sigma]\!]_\eta^! \to [\![\rho]\!]_\eta^?|$. Let $\sigma' \to \rho' \in \bar{\eta}(\sigma \to \rho)$, this implies $\sigma' \in \bar{\eta}(\sigma)$ and $\rho' \in \bar{\eta}(\rho)$. $[M] \in [\![\sigma \to \rho]\!]_\eta^\xi \Rightarrow B^+ \vdash M : \xi\sigma' \to \rho'$,

$[N]\in[\![\sigma]\!]^{\bar\xi}_\eta \Rightarrow B^+\vdash N:\bar\xi\sigma'$ and then $B^+\vdash MN:\xi\rho'$ by rule $(\to E)$, that is $[MN]\in[\![\rho]\!]^\xi_\eta$. For the converse, let $z\notin FV(M)\cup FV(B)$ and $z:\bar\xi\sigma'\in B^+$.

$$\begin{aligned}\forall[N]\in[\![\sigma]\!]^{\bar\xi}_\eta, [MN]\in[\![\rho]\!]^\xi_\eta &\Rightarrow B^+\vdash Mz:\xi\rho'\\ &\Rightarrow B^+\vdash \lambda z.Mz:\xi\sigma'\to\rho' \text{ by } (\to I)\\ &\Rightarrow [\lambda z.Mz]=\boldsymbol{\varepsilon}\cdot[M]\in[\![\sigma\to\rho]\!]^\xi_\eta.\end{aligned}$$

Condition (6). Let $\forall\varphi.\sigma'\in\bar\eta(\forall\varphi.\sigma)$.

$$\begin{aligned}[M]\in[\![\forall\varphi.\sigma]\!]^\xi_\eta &\Rightarrow B^+\vdash M:\xi\forall\varphi.\sigma'\\ &\Rightarrow B^+\vdash M:?\sigma'[\rho'/\varphi] \quad \text{where } \rho'\in\bar\eta(\rho) \text{ for all } \rho\in\mathscr{T} \text{ by } (\forall E?)\\ &\Rightarrow B^+\vdash M:?\bar\eta(\sigma[\rho/\varphi]) \quad \text{for all } \rho\in\mathscr{T} \text{ since } \sigma'[\rho'/\varphi]\in\bar\eta(\sigma[\rho/\varphi])\\ &\Rightarrow [M]\in\bigcap_{\rho\in\mathscr{T}}[\![\sigma[\rho/\varphi]]\!]^?_\eta.\end{aligned}$$

Condition (7). Let $\forall\varphi.\sigma'\in\bar\eta(\forall\varphi.\sigma)$.

$$\begin{aligned}[M]\in[\![\forall\varphi.\sigma]\!]^!_\eta &\Rightarrow B^+\vdash M:!\forall\varphi.\sigma'\\ &\Rightarrow B^+\vdash M:!\sigma'[\tau/\varphi] \quad \text{for all } \tau \text{ total by } (\forall E!)\\ &\Rightarrow B^+\vdash M:!\bar\eta(\sigma[\tau/\varphi]) \quad \text{for all } \tau \text{ total } (\tau \text{ total implies}\\ &\qquad\qquad \tau \text{ closed and therefore } \bar\eta(\tau)=[\tau])\\ &\Rightarrow [M]\in\bigcap_{\tau\,\text{total}}[\![\sigma[\tau/\varphi]]\!]^!_\eta.\end{aligned}$$

Condition (8).

$$\begin{aligned}[M]\in\bigcap_{I\in Type(\Lambda(\beta))}([\![\sigma]\!]_{\eta[I/\varphi]})^\xi &\Rightarrow [M]\in([\![\sigma]\!]_{\eta[I(\psi)/\varphi]})^\xi\\ &\qquad \text{where } \psi \text{ is bindable in } M \text{ with respect to } B\\ &\Rightarrow [M]\in([\![\sigma]\!]_{\eta^+})^\xi \quad \text{where } \eta^+=\eta[I(\psi)/\varphi]\\ &\Rightarrow B^+\vdash M:\xi\sigma' \quad \text{where } \sigma'\in\overline{\eta^+}(\sigma)\\ &\Rightarrow B^+\vdash M:\xi\forall\psi.\sigma' \quad \text{by } (\forall I)\\ &\Rightarrow B^+\vdash M:\xi\bar\eta(\forall\varphi.\sigma) \quad \text{by definition of } \eta^+\\ &\Rightarrow [M]\in[\![\forall\varphi.\sigma]\!]^\xi_\eta.\end{aligned}$$

Part (ii) follows immediately. □

THEOREM 3.6 (Completeness). (i) $\mathscr{M}(\beta)$, θ_0, $[\![\ ]\!]$, $\eta_0\models M:\xi\sigma\Rightarrow B\vdash M:\xi\sigma$.
(ii) $B\vdash M:\xi\sigma\Leftrightarrow B\models M:\xi\sigma$.

Proof. (i)

$$\begin{aligned}\mathscr{M}(\beta),\theta_0,[\![\ ]\!],\eta_0\models M:\xi\sigma &\Rightarrow [\![M]\!]^{\mathscr{M}(\beta)}_{\theta_0}\in[\![\sigma]\!]^\xi_{\eta_0}\\ &\Rightarrow [M]\in I(\sigma)^\xi\\ &\Rightarrow B^+\vdash M:\xi\sigma \text{ since } \sigma\in[\sigma]=\bar\eta_0(\sigma)\\ &\Rightarrow B\vdash M:\xi\sigma.\end{aligned}$$

Part (ii) follows immediately from (i) and Theorem 3.2. □

The inference system so defined, without other suitable rules, is not complete with respect to simple or $\mathscr{F}$-type interpretations since, for example, we have $\{x:!\sigma\to\rho\}\vdash\lambda y.xy:!(\forall\varphi.\varphi)\to\rho$ but $\{x:!\sigma\to\rho\}\nvdash x:!(\forall\varphi.\varphi)\to\rho$. Moreover, it is not complete with respect to total type interpretations, since, for example, $\{x:!\forall\varphi.\varphi\to\psi\}\vdash\lambda y.xy:!\tau\to\psi$ for all total τ but $\{x:!\forall\varphi.\varphi\to\psi\}\nvdash\lambda y.xy:!\forall\varphi.\varphi\to\psi$.

We can easily obtain two systems that are complete with respect to the simple and total semantics of types, while it is an open problem to find a system complete with respect to the $\mathscr{F}$-semantics.

DEFINITION 3.7 (Simple and total inference systems). (i) $B \vdash_s M : \xi\sigma$ if and only if this statement can be derived by the axioms and rules of Definition 1.3 together with rules (Eq) and

$$\text{(simple)} \qquad \frac{B \vdash \lambda z.Mz : \xi\sigma \to \rho}{B \vdash M : \xi\sigma \to \rho} \qquad \text{if } z \notin FV(M).$$

(ii) $B \vdash_T M : \xi\sigma$ if and only if this statement can be derived by the axioms and rules of Definition 1.3 together with rules (Eq) and

$$\text{(total)} \qquad \frac{B \vdash M : \xi\sigma[\tau/\varphi] \quad \text{for all } \tau \text{ total}}{B \vdash M : \xi\forall\varphi.\sigma}.$$

The soundness of rule (simple) with respect to the simple semantics and of rule (total) with respect to the total semantics is straightforward. Note that we must use transfinite induction for the system $\vdash_T$, since (total) is an infinitary rule.

To prove the completeness of these systems, we must modify the definitions of intervals and type interpretation on the term model, in an obvious way. These changes take into account the added rule to obtain a simple, or a total, type interpretation.

In the case of the simple semantics we need to introduce a relation $\approx_s$ on types. $\sigma \approx_s \rho$ says that σ and ρ can be assigned to the same set of terms in the system $\vdash_s$, i.e.,

$$\sigma \approx_s \rho \Leftrightarrow \forall B, M, \xi : B \vdash_s M : \xi\sigma \Leftrightarrow B \vdash_s M : \xi\rho.$$

$\approx_s$ extends $\approx$, since it is easy to verify that

$$B \vdash_s N : \xi\sigma \Leftrightarrow \exists M : M \twoheadrightarrow_\eta N \quad \text{and} \quad B \vdash M : \xi\sigma.$$

Actually, $\approx_s$ is a proper extension, since for example $\forall\varphi.\tau \to \varphi \approx_s \tau \to \forall\varphi.\varphi$ whenever τ is total. Let $[\sigma]_s$ denote the equivalence class of σ under $\approx_s$, i.e., $[\sigma]_s = \{\rho \mid \rho \approx_s \sigma\}$.

Let us now consider the system $\vdash_T$. Note that the addition of the infinitary rule (total) to the system $\vdash$ leaves unchanged the set of types that can be deduced for the (term) variables. More formally we have

$$\forall x, \sigma, \rho, \xi, \xi' : \{x : \xi\rho\} \vdash x : \xi'\sigma \Leftrightarrow \{x : \xi\rho\} \vdash_T x : \xi'\sigma.$$

Moreover, it is clear that the equivalence $\approx$ (and the corresponding equivalence for the system $\vdash_T$) is completely characterized by means of the deductions in which all subjects are variables. This is precisely stated by

$$(\forall B, M, \xi : B \vdash M : \xi\sigma \Leftrightarrow B \vdash M : \xi\rho)$$
$$\Leftrightarrow (\forall x, \xi : \{x : \xi\rho\} \vdash x : \xi\sigma \text{ and } \{x : \xi\sigma\} \vdash x : \xi\rho)$$

and the same implication where $\vdash_T$ replaces $\vdash$. From these facts it is easy to prove that $\approx$ characterizes the types that can be assigned to the same set of terms in the system $\vdash_T$, that is

$$\sigma \approx \rho \Leftrightarrow \forall B, M, \xi : B \vdash_T M : \xi\sigma \Leftrightarrow B \vdash_T M : \xi\rho.$$

So we simply define $[\sigma]_T = [\sigma]$.

DEFINITION 3.8. Let $i \in \{s, T\}$.

(i) $I^i(\sigma) = \{A \subseteq \Lambda(\beta) \mid \{[M] \mid B^+ \vdash_i M : !\sigma\} \subseteq A \subseteq \{[M] \mid B^+ \vdash_i M : ?\sigma\}\}$.

(ii) $Type^i(\Lambda(\beta)) = \{I^i(\sigma) \mid \sigma \in \mathscr{T}\}$.

(iii) Let η : Type Variables $\to$ $Type^i(\Lambda(\beta))$. Define:

$$\bar{\eta}(\sigma) = [\sigma[\sigma_1/\varphi_1, \cdots, \sigma_n/\varphi_n]_i]_i$$

where $\eta(\varphi_j) = I^i(\sigma_j)$ and $\varphi_1, \cdots, \varphi_n$ are the free variables of σ.

$$[\![\sigma]\!]^i_\eta = I^i(\rho) \quad \text{for some } \rho \in \bar{\eta}(\sigma).$$

(iv) $\eta_i(\varphi) = I^i(\varphi)$.

THEOREM 3.9 (Completeness for the simple and total semantics).

(i) $[\![\ ]\!]^s$ *is a simple type interpretation.*

(ii) $[\![\ ]\!]^T$ *is a total type interpretation.*

(iii) $B \vdash_s M : \xi\sigma \Leftrightarrow B \models_s M : \xi\sigma$.

(iv) $B \vdash_T M : \xi\sigma \Leftrightarrow B \models_T M : \xi\sigma$.

Proof. (i) As the proof of Lemma 3.5(i). Notice that

$$B^+ \vdash_s \lambda z.Mz : \xi\sigma' \to \rho',\ z \notin FV(M) \Rightarrow B^+ \vdash_s M : \xi\sigma' \to \rho' \text{ by (simple)}$$

$$\Rightarrow [M] \in ([\![\sigma \to \rho]\!]^s_\eta)^\xi.$$

The proof of part (ii) is that of 3.5(i). Note that

$$[M] \in \bigcap_{\tau \text{ total}} ([\![\sigma[\tau/\varphi]]\!]^T_\eta)^\xi \Rightarrow B^+ \vdash_T M : \xi\sigma'[\tau/\varphi]$$

where $\sigma' \in \bar{\eta}(\sigma)$ for all τ total (τ total implies $\bar{\eta}(\tau) = [\tau]$)

$$\Rightarrow B^+ \vdash_T M : \xi\forall\varphi.\sigma' \quad \text{by (total)}$$

$$\Rightarrow [M] \in ([\![\forall\varphi.\sigma]\!]^T_\eta)^\xi.$$

The proofs of parts (iii) and (iv) are that of Theorem 3.6(ii) using, respectively, (i) and (ii). □

Note that $\vdash_s$ is not complete with respect to the $\mathscr{F}$-semantics, since, for example, $\{x : \xi\forall\varphi.\varphi\} \vdash_s x : \xi\forall\varphi.\varphi$ but $\{x : \xi\forall\varphi.\varphi\} \not\vdash_s \lambda y.xy : \xi\forall\varphi.\varphi$.

Remark 3. Let us compare the present relations $\approx$ and $\approx_s$ with the containment $\subseteq$ between types introduced by Mitchell in [12]. Actually, Mitchell does not consider recursive or partial types, so the comparison is only done for total types built using $\to$ and $\forall$. It turns out that $\sigma \approx \rho$ ($\sigma \approx_s \rho$) if and only if $\sigma \subseteq \rho \subseteq \sigma$ is valid in all (simple) inference models.

4. Relations with standard inference systems. It is natural to compare the type inference system of Definition 1.3 with the system whose rules are obtained from the rules of 1.3 simply by erasing modalities in the assertions (obviously, $(?\Rightarrow!)$ and $(!\Rightarrow?)$ become meaningless and $(\forall E!)$ becomes a particular case of $(\forall E?)$). We write $B \vdash^* M : \sigma$ if this expression can be derived in the so-obtained system. Let $\bar{B} = \{x : \sigma \mid x : \xi\sigma \in B\}$. Clearly, $B \vdash M : \xi\sigma$ implies $\bar{B} \vdash^* M : \sigma$, since the erasure of modalities does not affect the validity of the derivations. The reverse does not hold, since $B \vdash^* M : \sigma$ does not imply that there exist suitable B' and ξ such that $\bar{B'} = B$ and $B' \vdash M : \xi\sigma$. Take, for example, $\{x : \forall\varphi.\varphi, y : \forall\varphi.\varphi \to \varphi,\ z : \varphi\} \vdash^* x(yz) : \varphi$. Moreover, it is clear that, in the particular case of total types, the interpretation of types as intervals coincides with the standard interpretation as sets. Using this fact and the completeness proved in § 3, it is easy to show that if B contains only total types and τ is total, then $B \vdash^* M : \tau$ implies $B' \vdash M : \xi\tau$ where $B' = \{x : \xi_x\sigma \mid x : \sigma \in B\}$ and the modalities ξ, ξ_x

are arbitrary. In fact, we have

$$B \vdash^* M : \tau \Rightarrow B \models M : \tau \quad \text{by soundness}$$
$$\Rightarrow B' \models M : \xi\tau \quad \text{since all involved types are total}$$
$$\Rightarrow B' \vdash M : \xi\tau \quad \text{by completeness.}$$

Note that the inference system $\vdash^*$ is different from the system presented in [8] for two reasons; in [8] other type constructors (such as $\times$, $\exists$) are defined and rule ($\sim$) is omitted.

5. Possible developments. An interesting point to investigate could be the connection between properties of subtyping (as introduced in [2]) and the approximation relation on types as intervals.

Let us consider two intervals I and J, say $I = |A, B|$ and $J = |C, E|$, such that $I \sqsubseteq J$. I represents an approximant of J, i.e., I is a supertype of J in the sense that all sets that are elements of J also belong to I (are included in B). This order relation on intervals seems to suggest that, exploiting the use of modalities in typing statements, we can recover some of the inheritance properties of types considered as sets of values. Indeed the following hold:

(1) $d \in J^{\xi}$ implies $d \in I^{?}$,

(2) $d \in I^{!}$ implies $d \in J^{!}$.

According to (1), if d belongs to the set of values C (or E) we infer that $d \in |A, B|^{?}$, i.e., d belongs to B (notice that $C \subseteq E \subseteq B$ by definition of $\sqsubseteq$). So values of C can inherit properties of the superset B. On the other hand, (2) allows us to pass from a less defined characterization to a more defined one. Assume, for example, that we have a function $f \in (I \Rightarrow |H, H|)^{!}$, which means that f belongs to the set $B \to H$. By the monotonicity of $\Rightarrow$, $I \Rightarrow |H, H|$ is less than $J \Rightarrow |H, H|$ and then, by property (2), f belongs to the set $E \to H$ where $E \subseteq B$. So the function f can be applied not only to values of the set B but also to values of the subset E. Again, values of a set inherit properties of a superset.

A related work is [9], where Martini models both explicit polymorphism and inheritance for subtypes using a modification of the interval model. Martini presents a sound model for an extension of the language FUN introduced in [3]; this extension supports also a general recursion operator for functions. To satisfy the requirement that the function type constructor is antimonotonic in the first argument, the subtype relation is not interpreted by the ordering on intervals.

Acknowledgments. The authors wish to thank the referee, Felice Cardone, and Mario Coppo for their helpful comments on a previous version of this paper.

REFERENCES

[1] H. BARENDREGT, *The Lambda Calculus: Its Syntax and Semantics*, North-Holland, Amsterdam, 1984 (revised edition).

[2] L. CARDELLI, *A semantics of multiple inheritance*, Inform. and Comput., 76 (1988), pp. 138–164.

[3] L. CARDELLI AND P. WEGNER, *On understanding types, data abstraction and polymorphism*, ACM Computing Surveys, 17 (1985), pp. 471–522.

[4] F. CARDONE AND M. COPPO, *Type inference with recursive types: syntax and semantics*, Inform. and Comput., to appear.

[5] R. CARTWRIGHT, *Types as intervals*, Conference Record of the 12th Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1984, pp. 22–36.

[6] J. Y. GIRARD, *Three-valued logic and cut-elimination: the actual meaning of Takeuti's conjecture*, Dissertationes Mathematicae, Warszawa, Poland, 1976, pp. 1-31.

[7] R. HINDLEY, *The completeness theorem for typing λ-terms*, Theoret. Comput. Sci., 22 (1983), pp. 1-17.

[8] D. MACQUEEN, G. PLOTKIN, AND R. SETHI, *An ideal model for recursive polymorphic types*, Inform. and Control, 71 (1986), pp. 95-130.

[9] S. MARTINI, *Bounded quantifiers have intervals models*, ACM Symposium on LISP and Functional Programming, Association for Computing Machinery, New York, 1988, pp. 174-183.

[10] A. R. MEYER, *What is a model of the lambda calculus?*, Inform. and Control, 52 (1982), pp. 87-122.

[11] A. R. MEYER, J. C. MITCHELL, E. MOGGI, AND R. STATMAN, *Empty types in polymorphic lambda calculus*, Conference Record of the 14th Annual ACM Symposium on Principles of Programming Languages, Association for Computing Machinery, New York, 1987, pp. 253-262.

[12] J. C. MITCHELL, *Polymorphic type inference and containment*, Inform. and Comput., 76 (1988), pp. 211-249.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 659-665, August 1990

005

SHORTEST CIRCUIT COVERS AND POSTMAN TOURS IN GRAPHS WITH A NOWHERE ZERO 4-FLOW*

BILL JACKSON†

Abstract. Let G be a graph with a nowhere zero 4-flow. It is shown that the length of a shortest circuit cover of $E(G)$ is equal to the length of a shortest postman tour of $E(G)$. Using this result an efficient algorithm for constructing shortest circuit covers for graphs that possess two disjoint spanning trees is obtained. It is also deduced that if H is a $2m$-edge connected graph, $m \geqq 2$, then there exists a circuit cover of $E(H)$ of length at most $|E(H)| + \min\{|E(H)|/(2m+1), |V(H)|-1\}$ and that if G has a nowhere zero 4-flow, then there exists a circuit cover of $V(G)$ of length at most $2|V(G)|-2$. Finally, it is shown that the equivalence between shortest circuit covers and postman tours may be extended to binary matroids that possess a nowhere zero $\mathbb{Z}_2^2$-flow.

Key words. shortest circuit covers, Chinese postman problem

AMS(MOS) subject classifications. 05C70, 94C15, 05C38

1. Introduction and definitions. All graphs considered are finite and may contain loops and multiple edges. We shall adopt the terminology of [J] concerning *flows* in graphs. By a *network* we shall mean a graph G for which a positive rational number $w(e)$ is associated with each edge e of G. We shall use $w(G)$ to denote the sum of the weights of the edges of G. We shall consider a graph G as a network for which $w(e) = 1$ for each edge e of G. A *closed walk* in G is an alternating sequence of vertices and edges of G that starts and ends at the same vertex and is such that consecutive vertices and edges are incident. The *length* of a walk Q is the sum of the weights of the edges in the sequence representing Q and is denoted by $w(Q)$. A *postman tour* of G is a closed walk that includes every edge of G. A *circuit* of G is a closed walk such that all vertices other than the first vertex are distinct. A *circuit cover* (*vertex circuit cover*) of G is a set of circuits such that each edge (vertex) of G belongs to at least one circuit in the set. The *length* of a circuit cover S is the sum of the lengths of its circuits and is denoted by $w(S)$. We shall use $p(G)$, $c(G)$, $cv(G)$ to denote the lengths of shortest postman tour, shortest circuit cover, and shortest circuit vertex cover, respectively.

Itai and Rodeh [IR] pointed out that, for any bridgeless graph G, $p(G) \leqq c(G)$, and gave the Petersen graph as an example for which strict inequality occurs. Since there exists an efficient algorithm for determining $p(G)$ and, as yet, no such algorithm for $c(G)$, it seems useful to describe families of graphs G for which $p(G) = c(G)$. In [BJJ] and [GFl] it is shown that $p(G) = c(G)$ for all planar bridgeless graphs G. The main purpose of this paper is to generalise this result by showing that $p(G) = c(G)$ for all networks G that have a nowhere zero 4-flow (note that all planar bridgeless graphs have a nowhere zero 4-flow by the 4-colour theorem [AH]). Unfortunately it is NP-complete to decide whether a given graph has a nowhere zero 4-flow (for 3-regular graphs it is equivalent to decide if the graph is 3-edge colourable and this decision problem is NP-complete by [Ho]). However, by specialising to the family of networks that possess two disjoint spanning trees (which have a nowhere zero 4-flow by [J]) we

* Received by the editors December 19, 1988; accepted for publication (in revised form) October 4, 1989. This work was carried out while the author was visiting the Department of Mathematics and Statistics, University of Auckland, New Zealand.

† Department of Mathematical Sciences, Goldsmiths' College, London SE14 6NW, United Kingdom.

obtain a large family of networks for which there exists an efficient algorithm $O(|V^3|+|E|)$, to construct a shortest circuit cover.

Several authors, [IR], [ILPR], [BJJ], [AT], [Fr], [F] have obtained upper bounds on the lengths of shortest circuit covers in graphs and networks. (Note that bounds for graphs may be applied to networks with integer weights, by replacing each edge e of a network by a path of length $w(e)$. This may be extended to networks with positive rational weights by suitable scaling.) A summary of the strongest results (from [BJJ], [AT], [Fr], [F]) is the following.

THEOREM 1.1. *Let G be a 2-edge connected network and $T(G)$ denote the maximum weight of a spanning tree of G. Then*

$$c(G) \leqq w(G) + \min\{2w(G)/3, 5T(G)/4\}.$$

In addition, for a 2-edge connected graph G:

(i) [AT] *gives a $O(|V|^2+|E|)$ algorithm for constructing a circuit cover of length at most $|E|+\min\{2|E|/3, 7(|V|-1)/3\}$.*

(ii) $c(G) \leqq |E|+|V|-1$ *if G has two edge-disjoint spanning trees* [IR], *or if G is planar* [Fr].

(iii) $c(G) \leqq 4|E|/3$ *if G has a nowhere zero 4-flow* [BJJ].

(iv) $c(G) \leqq 8|E|/5$ *if G has a nowhere zero 5-flow* [JRT].

(v) $cv(G) \leqq 50(|V|-1)/23$ *in general, and $cv(G) \leqq 2(|V|-1)$ if G is planar* [Fr].

In §§ 4 and 5 of this paper we shall improve the above results for the case of graphs that have a nowhere zero 4-flow.

Generalisations of the shortest circuit cover problem for binary matroids have been considered in [T], [JT], [JRT]. In § 6 of this paper we shall extend our main result to binary matroids that have a nowhere zero $\mathbb{Z}_2^2$-flow.

2. Preliminary lemmas. Let $G=(V, E)$ be a graph. We shall say $S \subset E$ is a $\mathbb{Z}_2$-*cycle* of G if each $v \in V$ is incident with an even number of edges of S. The *cycle space* of G is the vector space formed by the set of all $\mathbb{Z}_2$-cycles of G over $\mathbb{Z}_2$, using the binary operation symmetric difference (which we denote by Δ).

It was shown in [Tu] that, for each positive integer k, it is equivalent for a graph G to have a nowhere zero k-flow or a nowhere zero Γ-flow for any abelian group Γ of order k. We shall use the particular cases $k=2$ or 4.

LEMMA 2.1. *The following statements are equivalent.*

(1) *G has a nowhere zero 2-flow.*

(2) *G has a nowhere $\mathbb{Z}_2$-flow.*

(3) *$E(G)$ can be decomposed into disjoint circuits.*

(4) *$E(G)$ is a $\mathbb{Z}_2$-cycle.*

Furthermore, any incidence of (i) can be converted to an incidence of (j) in $O(|E|)$ time, $1 \leqq i < j \leqq 4$.

Proof. The equivalence follows easily. Flows satisfying (1) or (2) may be constructed using Euler tours of the components of G. To construct (3) we may greedily remove circuits from G. □

LEMMA 2.2. *The following statements are equivalent.*

(1) *G has a nowhere zero 4-flow.*

(2) *G has a nowhere zero $\mathbb{Z}_2^2$-flow.*

(3) *$E(G)$ is the union of two $\mathbb{Z}_2$-cycles.*

Furthermore, any incidence of (i) can be converted to an incidence of (j) in $O(|E|)$ time, $1 \leqq i < j \leqq 3$.

Proof. (1)⇒(2). Suppose $\varnothing$ is a nowhere zero 4-flow for G. Let $M_i = \{e \in E \mid \varnothing(e) \equiv i \pmod 2\}$ for $i=1$ or 2. Then M_1 is a $\mathbb{Z}_2$-cycle and hence by Lemma

2.1 we may construct a $\mathbb{Z}_2$-flow $\varnothing_1$ and a 2-flow $\varnothing_2$ for G such that $\varnothing_1(e) = 1 = \pm\varnothing_2(e)$ for $e \in M_1$ and $\varnothing_1(e) = 0 = \varnothing_2(e)$ for $e \in M_2$. Thus $\varnothing_3 = \varnothing + \varnothing_2$ is a flow for G with $\varnothing_3(e) \in \{0, \pm 2, \pm 4\}$ for all $e \in E$. Putting $\varnothing_4(e) = O$ if $\varnothing_3(e) \equiv O(\text{mod } 4)$ and $\varnothing_4(e) = 1$ if $\varnothing_3(e) \equiv 2(\text{mod } 4)$ we obtain a $\mathbb{Z}_2$-flow $\varnothing_4$ for G such that $\varnothing_4(e) = 1$ for $e \in M_2$. Thus $\varnothing_5 = (\varnothing_1, \varnothing_4)$ is the required nowhere zero $\mathbb{Z}_2^2$-flow for G.

(2)$\Rightarrow$(3). Immediate by Lemma 2.1.

(3)$\Rightarrow$(1). Let M_1 and M_2 be two $\mathbb{Z}_2$-cycles that cover E. By Lemma 2.1, we may construct 2-flows $\varnothing_i$ for G, $i = 1$ or 2, such that $\varnothing_i(e) = \pm 1$ for $e \in M_i$ and $\varnothing_i(e) = 0$ for $e \in E - M_i$. Putting $\varnothing = \varnothing_1 + 2\varnothing_2$ we obtain the required nowhere zero 4-flow for G.

The final assertion of the lemma follows, since our proofs are constructive and require at most $O(|E|)$ time. □

Following Jaeger [J], we shall say that an *F_4-flow* for G is a flow satisfying either Lemma 2.2.1 or 2.2.2. Our final lemma summarizes some well-known results on shortest postman tours of networks.

LEMMA 2.3. *Let G be a connected network and P be a shortest postman tour of G. Let F be the set of edges that are traversed more than once by P. Then*:

(1) *Each edge of F is traversed exactly twice by P.*

(2) *$E(G) - F$ is a $\mathbb{Z}_2$-cycle of G.*

(3) *$w(P) = w(G) + w(F)$.*

(4) *F contains no $\mathbb{Z}_2$-cycles of G.*

(5) *$w(F) \leqq t(G)$, where $t(G)$ denotes the minimum weight of a spanning tree of G.* □

3. Shortest circuit covers for networks that have an F_4-flow.

THEOREM 3.1. *Let G be a connected network that has an F_4-flow. Then $c(G) = p(G)$. Furthermore, if we are given an F_4-flow for G, then we can construct a shortest circuit cover for G in $O(|V|^3 + |E|)$ time.*

Proof. By Lemma 2.2 we can construct two $\mathbb{Z}_2$-cycles X_1 and X_2 of G such that $E(G) = X_1 \cup X_2$. Let P be a shortest postman tour of G, and F the set of edges traversed twice by P. By Lemma 2.3, $X = E(G) - F$ is a $\mathbb{Z}_2$-cycle of G. Put $D_1 = X_1 \Delta X$, $D_2 = X_2 \Delta X$, and $D_3 = X_1 \Delta X_2 \Delta X$. Then each D_i is a $\mathbb{Z}_2$-cycle of G, each edge of X belongs to exactly one D_i, and each edge of $F = E(G) - X$ belongs to exactly two D_i's, $1 \leqq i \leqq 3$. By Lemma 2.1, each D_i has a decomposition S_i into disjoint circuits. Putting $S = S_1 \cup S_2 \cup S_3$, we obtain a circuit cover S for G with

$$w(S) = w(G) + w(F) = w(P).$$

Since $c(G) \geqq p(G)$ by [IR] we have

$$c(G) = w(S) = w(P) = p(G).$$

Now suppose that we are given on F_4-flow $\varnothing$ for G. We may construct a shortest postman tour P for G in $O(|V|^3)$ time by [L]. Since X_1, X_2, X, $\{D_1, D_2, D_3\}$, $\{S_1, S_2, S_3\}$, and S can all be constructed from $\varnothing$ and P in $O(|E|)$ time, we obtain an $O(|V|^3 + |E|)$ algorithm for constructing S. □

COROLLARY 3.2. *Let G be a 2-edge connected network which has no subgraph contractible to $K_{3,3}$. Then $c(G) = p(G)$.*

Proof. It follows from [WW, Cor. 2] that G has an F_4-flow. □

The proof of [WW, Cor. 2] uses the 4-colour theorem. If we apply the 4-colour theorem directly, instead of using [WW, Cor. 2], we obtain the weaker result given in the following corollary.

COROLLARY 3.3. ([BJJ], [GFl]). *Let G be a 2-edge connected planar network. Then $c(G)=p(G)$.*

Proof. The 4-colour theorem [AH] implies that G has an F_4-flow. □

Remark 3.4. The proof of Corollary 3.3 given in [BJJ], [GFl] used a result of Fleischner [Fl] rather than the 4-colour theorem. Since Fleischner's proof is essentially constructive, it is conceivable that it could give rise to an efficient algorithm for constructing a shortest circuit cover of planar networks.

COROLLARY 3.5. *Let G be a network that has two disjoint spanning trees. Then $c(G)=p(G)$. Moreover a shortest circuit cover for G can be constructed in $O(|V|^3+|E|)$ time.*

Proof. It follows from [J] that G has an F_4-flow, and hence $c(G)=p(G)$ by Theorem 3.1. Furthermore, we can construct two disjoint spanning trees T_1 and T_2 for G (or decide that two such trees do not exist) in $O(|V|^2)$ time by [S]. Using T_1 and T_2 we may construct a nowhere zero $\mathbb{Z}_2^2$-flow for G in $O(|E|)$ time by [ILPR]. The corollary now follows by applying Theorem 3.1. □

4. Bounds on $c(G)$. Let $t(G)$ denote the minimum weight of a spanning tree of a connected network G.

THEOREM 4.1. *Let G be a connected network with an F_4-flow. Then $c(G)=p(G)\leqq w(G)+\min\{w(G)/3, t(G)\}$.*

Proof. The proof follows from Theorem 3.1, Lemma 2.3, and [BJJ, Prop. 2].

THEOREM 4.2. *Let G be a $2m$-edge connected network, $m\geqq 2$. Then $c(G)=p(G)\leqq w(G)+\min\{w(G)/(2m+1), t(G)\}$.*

Proof. Since G is 4-edge connected, it follows from [J] that G has an F_4-flow. The (in)equality $c(G)=p(G)\leqq w(G)+t(G)$ now follows from Theorem 4.1. It only remains to show

$$p(G)\leqq w(G)+w(G)/(2m+1). \tag{1}$$

We proceed by induction on $|E(G)|$, using an idea from the proof of [BJJ, Lem. 3.2]. First suppose that G has a vertex v of degree at least $2m+2$. Using [M] we may choose neighbours v_1 and v_2 of v and edges e_i, $i=1$ or 2, incident with v and v_i such that the graph H_1 obtained from $G-\{e_1, e_2]\}$ by adding a new edge e_{12} between v_1 and v_2 is $2m$-edge connected (we shall say that H_1 is obtained by *splitting* e_1 and e_2 from v). Putting $w(e_{12})=w(e_1)+w(e_2)$ and applying the inductive hypothesis to H_1, we deduce that (1) holds for G, since a shortest postman tour of H_1 gives rise to a postman tour of G with the same length.

Next, suppose G has a vertex v of degree $2m$. Again using [M] we may partition the edges incident with v into pairs such that the graph H_2, obtained by successively splitting each pair of edges away from v and then deleting the remaining isolated vertex v, is $2m$-edge connected. Applying induction to H_2 we again deduce that (1) holds for G.

We may now assume that G is $(2m+1)$-regular. It follows from results of [E] that G has a set of 1-factors S such that each edge of G belongs to the same number of 1-factors in S. Thus, if F is a 1-factor of S of minimum weight, then $w(F)\leqq w(G)/(2m+1)$. Choosing a postman tour P of G that traverses each edge of F twice and every other edge of G once, we have

$$w(P)\leqq w(G)+w(F)\leqq w(G)+w(G)/(2m+1).$$

Thus (1) holds for G. □

Remark 4.3. It was conjectured in [IR] that $c(G)\leqq|E|+|V|-1$ for all bridgeless graphs G. This conjecture was verified for graphs with two disjoint spanning trees in

[IR] and for planar graphs in [Fr]. Theorem 4.1 gives the common generalization that the conjecture holds for all graphs with an F_4-flow. It also suggests the following conjecture.

CONJECTURE 4.4. For every bridgeless network G, $c(G) \leqq w(G) + t(G)$.

5. Bounds on $cv(G)$. It was conjectured in [BJJ] that $cv(G) \leqq 2(|V|-1)$ for all 2-connected graphs G. The weaker bound $cv(G) \leqq 50(|V|-1)/23$ was established in [Fr], and the conjecture itself was verified for planar graphs. We shall extend the planar result by verifying the conjecture for graphs which have an F_4-flow.

THEOREM 5.1. *Let G be a graph without isolated vertices, which has an F_4-flow. Then $cv(G) \leqq 2(|V|-1)$.*

Proof. We proceed by contradiction. Suppose the theorem is false and choose a counterexample G with as few edges as possible. Clearly G is connected. Let U be the set of vertices of degree two in G. If $U = V$, then it can easily be seen that $cv(G) = |V|$, and thus $U \neq V$.

Suppose $G - U$ contains a circuit C. Choose a nowhere zero $\mathbb{Z}_2^2$-flow $\varnothing_1$ for G, and an edge e_0 of C. Let $\varnothing_2$ be the $\mathbb{Z}_2^2$-flow for G defined by $\varnothing_2(e) = \varnothing_1(e_0)$ for $e \in E(C)$ and $\varnothing_2(e) = (0, 0)$ for $e \in E - E(C)$. Let $\varnothing_3 = \varnothing_1 + \varnothing_2$, and $M = \{e \in E \mid \varnothing_3(e) = (0, 0)\}$. Then $e_0 \in M$. Since $G - M$ has no isolated vertices, has the F_4-flow $\varnothing_3$, and has fewer edges than G, we deduce that $cv(G - M) \leqq 2(|V|-1)$. Since any circuit vertex cover of $G - M$ is also a circuit vertex cover of G, we contradict the choice of G. Hence $G - U$ has no circuits.

Let H be the graph homeomorphic to G and without vertices of degree two. Thus $|E(H)| = |E| - |U|$ and $|V(H)| = |V| - |U|$. Clearly an F_4-flow in G gives rise to an F_4-flow in H. Associate a weight $w(e)$ with each $e \in E(H)$ by letting $w(e)$ be the length of the path in G that corresponds to e. Thus $w(H) = |E|$. Let $F = \{e \in E(H) \mid w(e) = 1\}$. Since $G - U$ is a forest, we can choose a spanning tree T for H such that $F \subset E(T)$. Then each edge of $E(H) - E(T)$ has weight at least two and thus,

$$\begin{aligned} w(T) &\leqq w(H) - 2(|E(H)| - |E(T)|) \\ &= w(H) - 2(|E(H)| - |V(H)| + 1) \\ &= 2(|V|-1) - |E|. \end{aligned}$$

Using Theorem 4.1 we have

$$c(H) \leqq w(H) + t(H) \leqq |E| + 2(|V|-1) - |E| = 2(|V|-1).$$

Since any circuit cover of H induces a circuit cover of G of the same length, we have $cv(G) \leqq c(H) \leqq 2(|V|-1)$. This contradicts the choice of G and completes the proof of the theorem. □

6. Minimum weight cycle covers in binary matroids. We define a *binary matroid* to be an ordered pair $M = (E, N)$, where E is a finite set and N is a subspace of 2^E considered as a vector space over $\mathbb{Z}_2$ using the binary operation Δ. We shall refer to N as the *cycle space* of M and to the elements of N as *cycles* of M. A *cycle cover* for M is a set of cycles whose union is E. We shall say that M is *weighted* if for each $e \in E$ there is associated a positive number $w(e)$, called the *weight* of e. For $F \subset E$, the *weight* of F is the sum of the weights of its elements. The *weight* of a cycle cover is the sum of the weights of its cycles.

The *cycle matroid* of a graph $G = (V, E)$ is the binary matroid $M = (E, N)$ defined by the cycle space N of G. Thus the cycles of M are the $\mathbb{Z}_2$-cycles of G.

LEMMA 6.1. Let $M=(E, N)$ be a binary matroid, X be a maximum weight cycle in M, and S be a minimum weight cycle cover for M. Then

$$w(S) \geqq 2w(E)-w(X).$$

Proof. Let $Z=\Delta_{Y\in S}\, Y$. Since S is a cycle cover, we have

$$w(S) \geqq w(Z)+2w(E-Z)=2w(E)-w(Z).$$

Furthermore, since $Z \in N$, we have $w(Z) \leqq w(X)$. □

Remark 6.2. If G is a connected network and X is a maximum weight $\mathbb{Z}_2$-cycle in G, then $p(G)=2w(E)-w(X)$. Thus Lemma 6.1 can be considered a generalisation of the result from [IR] that $c(G) \geqq p(G)$.

A *co-cycle* of the binary matroid $M=(E, N)$ is a subset $Y \subset E$ such that $|Y \cap X|$ is even for all $X \in N$. The set of all co-cycles N^* is a subspace of 2^E called the *co-cycle space* of M. The binary material $M^*=(E, N^*)$ is the *dual matroid* to M. A $\mathbb{Z}_2^k$-flow in M is a mapping $\varnothing: E \to \mathbb{Z}_2^k$ such that $\sum_{e\in Y} \varnothing(e)=0$ for all $Y \in N^*$. We shall use the following elementary result on $\mathbb{Z}_2^k$-flows to extend Theorem 3.1 from networks to binary matroids.

LEMMA 6.3. *Let $M=(E, N)$ be a binary matroid. Then M has a nowhere zero $\mathbb{Z}_2^k$-flow if and only if E is the union of k cycles in N.* □

THEOREM 6.4. *Let $M=(E, N)$ be a binary matroid that has a nowhere zero $\mathbb{Z}_2^2$-flow. Let X be a maximum weight cycle in M. Then the minimum weight for a cycle cover for M is $2w(E)-w(X)$.*

Proof. By Lemma 6.3 there exists $X_1, X_2 \in N$ such that $E=X_1 \cup X_2$. Then $S=\{X_1 \Delta X, X_2 \Delta X, X_1 \Delta X_2 \Delta X\}$ is a cycle cover for M such that $w(S)+w(X)=2w(E)$. Using Lemma 6.1 we deduce that S is a minimum weight cycle cover for M and $w(S)=2w(E)-w(X)$. □

The *co-cycles* or *edge-cuts* of a graph $G=(V, E)$ are the co-cycles of the cycle matroid of G. These are the subsets of E consisting of all edges joining U to $V-U$ for each $U \subset V$. The *co-cycle matroid* of G is the dual matroid to the cycle matroid of G. Applying Theorem 6.4 to the co-cycle matroid of a graph we obtain Corollary 6.5.

COROLLARY 6.5. *Let $G=(V, E)$ be a 4-colourable network and r be the maximum weight of a co-cycle of G. Then the minimum weight of a co-cycle cover of E is $2w(E)-r$.*

Proof. The co-cycle matroid of G has a $\mathbb{Z}_2^2$-flow if and only if G is 4-colourable. □

Remark 6.6. Unfortunately the problem of determining the maximum weight of a co-cycle in a graph, and thus the more general problem of determining the maximum weight of a cycle in a binary matroid, is NP-hard [GJS]. This remains valid for loopless graphs of maximum degree three (which are necessarily 4-colourable) by [Y], and hence the maximum weight cycle problem is NP-hard even for binary matroids that have a nowhere zero $\mathbb{Z}_2^2$-flow.It follows that Lemma 6.1, Theorem 6.4, and Corollary 6.5 are not as readily applicable to binary matroids as graphs. Since, however, polynomial algorithms exist for constructing maximum weight co-cycles in planar graphs [H], [OD], and more generally in graphs with no K_5-minor [B], we may apply Corollary 6.5 in these cases to give the following.

THEOREM 6.7. *Let G be a graph with no loops and no subgraph contractible to K_5 and let r be the maximum weight of a co-cycle in G. Then the minimum weight of a co-cycle cover of E is $2w(E)-r$. Furthermore, r may be determined in $O(|V|^5)$ time.*

Proof. It follows from the 4-colour theorem [AH] and a result of Wagner [W] that G is 4-colourable. Applying Corollary 6.5 we deduce that the minimum weight

of a co-cycle cover of G is $2w(E) - r$. The assertion that r may be calculated in $O(|V|^5)$ time follows from [B].

Acknowledgment. I thank François Jaeger for suggestions and comments that gave rise to § 6 of this paper.

REFERENCES

[AH] K. Appel and W. Haken, *Every planar map is 4-colourable*, Illinois J. Math., 21 (1977), pp. 429-567.

[AT] N. Alon and M. Tarsi, *Covering multigraphs by simple circuits*, SIAM J. Algebraic Discrete Methods, 6 (1985) pp. 345-350.

[B] F. Barahona, *The max-cut problem on graphs not contractible to K_5*, Oper. Res. Lett., 3 (1983), pp. 107-111.

[BJJ] J. C. Bermond, B. Jackson, and F. Jaeger, *Shortest coverings of graphs with cycles*, J. Combin. Theory Ser. (B) 35 (1983), pp. 297-308.

[E] J. Edmonds, *Maximum matching and a polyhedron with* 0, 1 *vertices*, J. Res. Nat. Bur. Standards, 69B (1965), pp. 125-130.

[GFl] M. Guan and H. Fleischner, *On the cycle covering problem for planar graphs*, Ars Combin., 20 (1985), pp. 61-68.

[GJS] M. R. Garey, D. S. Johnson, and L. Stockmeyer, *Some simplified* NP*-complete graph problems*, Theoret. Comput. Sci., 1 (1976), pp. 237-267.

[F] G. Fan, *Covering weighted graphs by even subgraphs*, Res. Report, University of Waterloo, Department of Combinatorics and Optimization, 1988.

[Fl] H. Fleischner, *Eulersche Linien und Kreisuberdeckungen, die vorgegebene Durchgänge in den Knoten vermeiden*, J. Combin. Theory Ser. (B), 29 (1980), pp. 145-167.

[Fr] P. Fraisse, *Cycle covering in bridgeless graphs*, J. Combin. Theory Ser. (B), 39 (1985), pp. 146-152.

[H] F. O. Hadlock, *Finding a maximum cut of a planar graph in polynomial time*, SIAM J. Comput., 4 (1975), pp. 221-225.

[Ho] I. Holyer, *The* NP*-completeness of edge-colouring*, SIAM J. Comput., 10 (1981), pp. 718-720.

[ILPR] A. Itai, R. J. Lipton, C. H. Papadimitrou, and M. Rodeh, *Covering graphs by simple circuits*, SIAM J. Comput., 10 (1981), pp. 746-754.

[IR] A. Itai and M. Rodeh, *Covering a graph by circuits*, in Automata, Languages and Programming, Lecture Notes in Computer Science, 10 (1981), pp. 746-754.

[J] F. Jaeger, *Flows and generalized colouring theorems in graphs*, J. Combin. Theory Ser. (B), 26 (1979), pp. 205-216.

[JRT] V. Jamshy, A. Raspaud, and M. Tarsi, *Short circuit covers for regular matroids*, J. Combin. Theory Ser. (B), 43 (1987), pp. 354-357.

[JT] V. Jamshy and M. Tarsi, *Cycle covering of binary matroids*, J. Combin. Theory Ser. (B), 46 (1989), pp. 154-161.

[L] E. L. Lawler, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart, Winston, New York, 1976.

[M] W. Mader, *A reduction method for edge-connectivity in graphs*, Advances in Graph Theory, Annals of Discrete Mathematics 3, North Holland, Amsterdam, 1978, pp. 145-164.

[OD] G. I. Orlova and Y. G. Dorfman, *Finding the maximum cut in a graph*, Engrg. Cybernetics, 10 (1972), pp. 502-506.

[S] Y. Shiloach, *Edge-disjoint branchings in directed multigraphs*, Inform. Process. Lett., 8 (1979), pp. 24-27.

[T] M. Tarsi, *Nowhere zero flow and circuit covering in regular matroids*, J. Combin. Theory Ser. (B), 39 (1985), pp. 346-352.

[Tu] W. T. Tutte, *A contribution to the theory of chromatic polynomials*, Canad. J. Math., 6 (1954), pp. 80-91.

[W] K. Wagner, *Über eine Eigenschaft der ebenen Komplexe*, Math. Ann., 114 (1937), pp. 570-590.

[WW] P. N. Walton and D. J. A. Welsh, *On the chromatic number of binary matroids*, Mathematika, 27 (1980), pp. 1-9.

[Y] M. Yannakakis, *Node- and edge-deletion* NP*-complete problems*, Proc. 10th Ann. ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1987, pp. 253-264.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 666-672, August 1990

006

GREEDY MATCHING ON THE LINE*

ALAN FRIEZE,† COLIN McDIARMID,‡ AND BRUCE REED§

Abstract. The problem of finding a perfect matching of small total length in a complete graph whose vertices are points in the interval [0, 1] is considered. The greedy heuristic for this problem repeatedly picks the two closest unmatched points x and y, and adds the edge xy to the matching. It is shown that if $2n$ points are randomly chosen uniformly in [0, 1], then the expected length of the matching given by the greedy algorithm is $\theta(\log n)$. This compares unfavourably with the length of the shortest perfect matching, which is always less than 1.

Key words. greedy, matching, Euclidean, line, average case, worst case

AMS(MOS) subject classifications. 68Q25, 90C27, 68R10, 05C70, 60C05, 60D05

1. Introduction. We are interested in finding a perfect matching of small length on a set of points drawn from the interval [0, 1]. That is, given a set A of $2n$ numbers $\{x_1, \cdots, x_{2n}\}$, each of which is between 0 and 1, we want to partition A into n pairs $\{y_1, z_1\}, \{y_2, z_2\}, \cdots, \{y_n, z_n\}$ so that we minimize $\sum_{i=1}^{n} |y_i - z_i|$. We can solve this problem by reordering the elements of A so that $x_{(1)} \leqq x_{(2)} \leqq \cdots \leqq x_{(2n)}$ and then set $y_i = x_{(2i-1)}$ and $z_i = x_{(2i)}$. It is easy to see that this method always gives the optimal matching and, furthermore, the length of this matching is at most 1.

It is natural to ask how well the greedy approach performs in this simple setting. The greedy matching algorithm first finds distinct elements a, b in A such that $|a-b| = \min\{|c-d|: c, d \in A, c \neq d\}$ and selects $\{a, b\}$ as one pair of the partition. The remaining pairs are found by using the same procedure on $A - \{a, b\}$.

It is not difficult to see that the greedy matching algorithm selects a matching of length at most $O(\ln(n))$ when applied to a set of $2n$ points. Also, one may construct examples to show that the worst-case weight of a greedy matching is $\Omega(\ln(n))$. Indeed, in § 5, we identify the worst-case behaviour rather precisely. (For related work see Avis [1] and Rheingold and Tarjan [4].) However, we are more interested in average-case behaviour. We prove that if $2n$ points are chosen at random from the uniform distribution on [0, 1], then the expected length of the resulting matching is $\Omega(\ln(n))$. This settles a question raised by Avis, Davis, and Steele in [2], where results are given on greedy Euclidean matching in d-dimensional spaces for $d \geqq 2$. (Note that the greedy algorithm may be of practical use when $d \geqq 2$, though it is only of theoretical interest in the case $d = 1$ considered here.)

Since this is our main result, we now state it again. Given n numbers $x_1, \cdots, x_n$ (n even) in the interval [0, 1], let $G[x_1, \cdots, x_n]$ denote the length of the corresponding greedy matching (break ties arbitrarily).

THEOREM. *Let $X_1, \cdots, X_n$ be n independent random variables, each uniformly distributed on* [0, 1]. *Then $E[G(X_1, \cdots, X_n)] \geqq \frac{1}{12} \ln(n)$ for n sufficiently large.*

2. Bags, sticks, and entropy. Rather than considering our points directly we shall focus on the distances between them. To this end, we begin with some definitions. Given an n-tuple $\mathbf{x} = (x_1, \cdots, x_n)$ of reals in [0, 1], let $x_{(1)} \leqq x_{(2)} \leqq \cdots \leqq x_{(n)}$ denote the numbers rearranged in nondecreasing order and let $z_k = x_{(k+1)} - x_{(k)}$ for $k = 0, \cdots, n$

* Received by the editors September 28, 1988; accepted for publication (in revised form) October 9, 1989.
† Department of Mathematics, Carnegie Mellon University, Pittsburgh, Pennsylvania 15213.
‡ Department of Statistics, Oxford University, Oxford, England.
§ Department of Combinatorics and Optimization, University of Waterloo, Waterloo, Canada.

where $x_{(0)} = 0$, $x_{(n+1)} = 1$. We shall refer to $z_1, \cdots, z_{n-1}$ as the (nearest neighbour) *sticks* and z_0, z_n as the *endsticks*. Let $L(\mathbf{x}) = [z_1, \cdots, z_{n-1}]$ and let $B(\mathbf{x})$ be the unordered (multi)set of these values. Thus $L(\mathbf{x})$ is the *l*ist of sticks and $B(\mathbf{x})$ the *b*ag of sticks. Note that the endsticks are not included here.

Now, let $X_1, \cdots, X_n$ be independently and identically distributed uniform on $[0, 1]$. We shall use two easy properties of the corresponding random sticks.

Property 1. Let B be any bag of $n-1$ sticks (considered distinct). Then, conditional on $B(\mathbf{X}) = B$ the distribution of the corresponding list $L(\mathbf{X})$ is uniform on the $(n-1)!$ orderings of the $n-1$ sticks in B.

This result is intuitively obvious, or see Feller [3], pp. 74–76.

Property 2. With probability $\to 1$ as $n \to \infty$,

$$\max\{Z_k : k = 0, \cdots, n\} \leqq 2 \ln(n)/n.$$

To check this result, note that for each k

$$\text{Prob}(Z_k > t) = (1-t)^n \quad \text{if } 0 < t < 1$$

(see, for example, Feller [3], p. 22). Thus

$$\text{Prob}(\max Z_k > t) \leqq (n+1)(1-t)^n \leqq (n+1)/n^2 \quad \text{if } t = 2\ln(n)/n.$$

To begin, we use the first property to obtain a lower bound on the conditional expected value $E(G[\mathbf{X}] \mid B[\mathbf{X}] = B)$ that depends on the length of the sticks in B. Then we point out that the second property ensures that this lower bound gives the desired result.

Thus, we now focus our attention on a fixed bag B of n sticks (where n is odd). By Property 1 above, the distribution of $G(\mathbf{X})$ conditional on $B(\mathbf{X}) = B$ is the same as that of the value CHOOSE(B) returned by the following randomized recursive algorithm.

Let x be a minimum element in B
if $|B| = 1$ *then return* x
else
with probability $2/n$ choose y uniformly from $B - \{x\}$ and *return* $x +$ CHOOSE $(B - \{x, y\})$ (this corresponds to x being the leftmost or rightmost stick, with neighbour y)
with probability $(n-2)/n$ choose y uniformly from $B - \{x\}$ and z uniformly from $B - \{x, y\}$ and *return* $x +$ CHOOSE $(B - \{x, y, z\} \cup \{x + y + z\})$ (this corresponds to x having left neighbour y and right neighbour z)

Now let

$$F(B) = E[\text{CHOOSE}(B)] = E[G(\mathbf{X}) \mid B(\mathbf{X}) = B].$$

By considering the algorithm CHOOSE (B) we obtain a recurrence for $F(B)$. Let $B = \{a_1, a_2, \cdots, a_n\}$ be a bag of n sticks where n is odd, $n > 1$, and a_1 is a minimum element. Then

$$F(B) = a_1 + \frac{2}{n}\frac{1}{n-1}\sum_{j=2}^{n} F(B - \{a_1, a_j\}) + \frac{n-2}{n}\frac{2}{(n-1)(n-2)}\sum_{j=2}^{n-1}\sum_{k=j+1}^{n} F(B - \{a_1, a_j, a_k\} \cup \{a_1 + a_j + a_k\}).$$

This recurrence is the key to our analysis. It will allow us to prove a lower bound on $F(B)$ in terms of the entropy of the stick lengths.

3. **A lemma.**

LEMMA. *Let* $A=\{a_1, \cdots, a_N\}$ *be an odd cardinality set of positive numbers that sum to* 1. *Let*

$$H(A)=\alpha \sum_{i=1}^{N} a_i \ln\left(\frac{1}{a_i}\right)-\beta \sum_{i=2}^{N} \frac{1}{i}-2\alpha \sum_{i=2}^{N} \frac{\ln(i)}{i^2},$$

where $\alpha=\dfrac{1}{\ln(12)}\approx 0.40243$ *and* $\beta=\alpha \ln(24/11)\approx 0.31396$. *Then* $F(A)\geqq H(A)$.

Proof. We note that if $N=1$, then $H(A)=0$ and $F(A)=1$, so the inequality holds. We shall assume it holds for $N<n$ (where $n>1$) and prove it for $N=n$. Furthermore, we may assume that a_1 is a minimal element of A. Then, as we noted earlier:

$$F(A)=a_1+\frac{2}{n(n-1)} \sum_{j=2}^{n} F(A-\{a_1, a_j\})$$

$$+\frac{2}{n(n-1)} \sum_{j=2}^{n-1} \sum_{k=j+1}^{n} F(A-\{a_1, a_j, a_k\}\cup\{a_1+a_j+a_k\}).$$

By the induction hypothesis,

$$F(A)\geqq a_1+\frac{2}{n(n-1)} \sum_{j=2}^{n} H(A-\{a_1, a_j\})$$

$$+\frac{2}{n(n-1)} \sum_{j=2}^{n-1} \sum_{k=j+1}^{n} H(A-\{a_1, a_j, a_k\}\cup\{a_1+a_j+a_k\}).$$

By the definition of H,

$$F(A)\geqq H(A)+a_1+\frac{\beta}{n}+\frac{\beta}{n-1}+\frac{2\alpha \ln(n)}{n^2}+\frac{2\alpha \ln(n-1)}{(n-1)^2}$$

$$+\frac{2\alpha}{n(n-1)} \sum_{j=2}^{n} \left(-a_j \ln\left(\frac{1}{a_j}\right)-a_1 \ln\left(\frac{1}{a_1}\right)\right)$$

$$+\frac{2\alpha}{n(n-1)} \sum_{j=2}^{n-1} \sum_{k=j+1}^{n} \left((a_j+a_k+a_1) \ln\left(\frac{1}{a_j+a_k+a_1}\right)\right.$$

$$\left.-a_j \ln\left(\frac{1}{a_j}\right)-a_k \ln\left(\frac{1}{a_k}\right)-a_1 \ln\left(\frac{1}{a_1}\right)\right).$$

So,

$$F(A)\geqq H(A)+a_1+\frac{2\beta}{n}+\frac{2\alpha \ln(n)}{n^2}+\frac{2\alpha \ln(n-1)}{(n-1)^2}-\alpha a_1 \ln\left(\frac{1}{a_1}\right)$$

$$-\frac{2\alpha}{n(n-1)} \sum_{j=2}^{n} a_j \ln\left(\frac{1}{a_j}\right)$$

$$+\frac{2\alpha}{n(n-1)} \sum_{j=2}^{n-1} \sum_{k=j+1}^{n} \left\{(a_j+a_k+a_1) \ln\left(\frac{1}{a_j+a_k+a_1}\right)\right.$$

$$\left.-a_j \ln\left(\frac{1}{a_j}\right)-a_k \ln\left(\frac{1}{a_k}\right)\right\}.$$

We note that by the convexity of $x \ln(x)$, the single sum is at most $\ln(n-1)$. We shall show that, given that $a_1 = a$, the double sum is minimized when $a_2 = \cdots = a_n = (1-a)/(n-1) = b$, say.

CLAIM.

$$S(A) = \sum_{j=2}^{n-1} \sum_{k=j+1}^{n} \left\{ (a_j + a_k + a_1) \ln\left(\frac{1}{a_j + a_k + a_1}\right) - a_j \ln\left(\frac{1}{a_j}\right) - a_k \ln\left(\frac{1}{a_k}\right) \right\}$$
$$\geqq \frac{1}{2}(n-1)(n-2)\left((a+2b)\ln\left(\frac{1}{a+2b}\right) - 2b\ln\left(\frac{1}{b}\right)\right).$$

Proof of Claim. Suppose $a_2 \neq a_3$. Let $a_2' = a_3' = (a_2 + a_3)/2$ and let $a_i' = a_i$ for $i = 1, 4, 5, \cdots, n$. It will suffice to show that $S(A') < S(A)$. To this end let $f_k(x) = x \ln(x) - (x + a_1 + a_k)\ln(x + a_1 + a_k)$ for $k = 4, \cdots, n$.

Now,

$$S(A') - S(A) = \{a_2' \ln(a_2') + a_3' \ln(a_3') - a_2 \ln(a_2) - a_3 \ln(a_3)\}$$
$$+ \sum_{i=4}^{n} (f_i(a_2') + f_i(a_3') - f_i(a_2) - f_i(a_3)).$$

Here, the first term comes from the term in the double sum where $j = 2$ and $k = 3$, while the ith term in the sum comes from the terms in the double sum where $k = i$ and $j = 2$ or $j = 3$. Since the functions $x \ln(x)$ and $f_k(x)$ are strictly convex, we have $S(A') - S(A) < 0$ as required. □

Applying the claim, we see that

$$F(A) \geqq H(A) + a + \frac{2\beta}{n} + 2\alpha\frac{\ln(n)}{n^2} - \alpha a \ln\left(\frac{1}{a}\right)$$
$$+ \left(\frac{n-2}{n}\right)\alpha\left((a+2b)\ln\left(\frac{1}{a+2b}\right) - 2b\ln\left(\frac{1}{b}\right)\right)$$
$$\geqq H(A) + \frac{2\beta}{n} + 2\alpha\frac{\ln(n)}{n^2} - \frac{2\alpha}{n} a \ln\left(\frac{1}{a}\right)$$
$$+ \frac{(n-2)\alpha}{n}\left(\frac{a}{\alpha} + a\ln\left(\frac{a}{a+2b}\right) + 2b\ln\left(\frac{b}{a+2b}\right)\right).$$

Setting $r = b/a$ and noting that $a \ln(1/a) \leqq \ln(n)/n$ since $a \leqq 1/n$, we get

$$F(A) \geqq H(A) + \frac{2\beta}{n} + \left(\frac{n-2}{n}\right)\alpha b f(r),$$

where $f(r) = (1/\alpha r) + (1/r)\ln(1/(1+2r)) + 2\ln(r/(1+2r))$. Now $f'(r) = (1/r^2)(-(1/\alpha) + \ln(1+2r))$, so $f(r)$ is minimized when $\ln(1+2r) = 1/\alpha$ and thus $f(r) \geqq -2\ln(2 + (2/e^{1/\alpha} - 1))$. Hence

$$F(A) \geqq H(A) + \frac{2\beta}{n} - \left(\frac{n-2}{n}\right)2\alpha b \ln\left(2 + \frac{2}{e^{1/\alpha} - 1}\right).$$

Next we note that $b \leqq 1/(n-1)$, so

$$F(A) \geqq H(A) + \frac{1}{n}\left(2\beta - 2\alpha \ln\left(2 + \frac{2}{e^{1/\alpha} - 1}\right)\right).$$

Thus, by our choice of α and β, we have $F(A) \geqq H(A)$ as required.

4. Completing the proof of the Theorem. Consider n points $x_1, \cdots, x_n$ in $[0,1]$ (n even) such that each of the corresponding $(n+1)$ sticks (including endsticks) has length at most $d(<\frac{1}{2})$. Thus the corresponding bag B of $n-1$ sticks has sum of lengths $\sigma \geqq 1-2d$. From the lemma (by scaling by $1/\sigma$)

$$F(B) \geqq \sigma\left[\alpha \sum_{x\in B} \frac{x}{\sigma} \ln\left(\frac{\sigma}{x}\right) - \beta \sum_{i=2}^{n-1} 1/i - 2\alpha \sum_{i=2}^{n-1} \frac{\ln(i)}{i^2}\right]$$

$$\geqq \sigma\left[\alpha\left(\sum_{x\in B} \frac{x}{\sigma}\right) \ln\frac{1-2d}{d} - \beta \sum_{i=2}^{n-1} 1/i - 2\alpha \sum_{i=2}^{n-1} \frac{\ln(i)}{i^2}\right]$$

$$= (1-2d)\left[\alpha \ln\frac{1-2d}{d} - \beta \sum_{i=2}^{n-1} 1/i - 2\alpha \sum_{i=2}^{n-1} \frac{\ln(i)}{i^2}\right].$$

Now set $d = 2\ln(n)/n$ and use Property 2. We find that

$$E[G(X_1, \cdots, X_n)] \geqq (1+o(1))E[G(X_1, \cdots, X_n) \mid \max_{0\leqq j\leqq n} Z_j \leqq 2\ln(n)/n]$$

$$\geqq (1+o(1))((\alpha+o(1))\ln(n) - \beta\ln(n) + O(1))$$

$$= (\alpha - \beta + o(1))\ln(n) \approx (0.088)\ln(n).$$

This completes the proof of the theorem. □

5. Worst-case results. In this section we consider lists of points on which the greedy algorithm performs particularly badly.

For any nonnegative integer k consider the list $\mathbf{x}(k) = \{i/3^k : i = 0, 1, \cdots, 3^k\}$ of 3^k+1 points in $[0,1]$. For $k \geqq 1$ the greedy algorithm applied to $\mathbf{x}(k)$ can pick 3^{k-1} intervals of length 3^{-k}, namely, the intervals $[(3j+1)3^{-k}, (3j+2)3^{-k}]$ for $j = 0, 1, \cdots, 3^{k-1}-1$, and then be left with the points $\mathbf{x}(k-1)$. Hence $\mathbf{x}(k)$ has a greedy matching of weight $k/3+1$. It follows that for any even n, there is a list of n points that has a greedy matching of weight $\frac{1}{3}\lfloor \log_3(n-1)\rfloor + 1$.

On the other hand, we shall show that any greedy matching on a list of n points has weight at most $\frac{1}{3}\log(n-1)+1$. Thus for n of the form 3^k+1 the above examples are worst possible and for all other n we are not far off. From now on we shall just use $\log(x)$ to denote $\log_3(x)$.

PROPOSITION. *For a bag $A = \{a_1, \cdots, a_N\}$ of sticks (with N odd), let $W(A)$ be the length of the longest greedy matching of any n-tuple $\mathbf{x}$ such that the corresponding list $L(\mathbf{x})$ of sticks is a permutation of A. Then*

$$W(A) \leqq \frac{1}{3}\sum_{i=1}^{N} a_i \log\left(\frac{1}{a_i}\right) + \sum_{i=1}^{N} a_i.$$

Proof. We prove this by induction on the cardinality of A. If $|A| = 1$, then $W(A) = a_1$ and the theorem is true. So we suppose the theorem holds for $N < n$, where $n \geqq 3$, and prove it for $N = n$. Let $\mathbf{x}$ be a list of points such that $L(\mathbf{x})$ is a permutation of A and such that $G(\mathbf{x}) = W(A)$. We may assume that a_1 is the minimal element of A chosen in the first iteration when the greedy matching is applied to $\mathbf{x}$.

Case 1. a_1 is the rightmost or leftmost stick of $L(\mathbf{x})$. In this case, let a_2 be the neighbour of a_1 in $L(\mathbf{x})$. Then

$$W(A) = G(\mathbf{x}) \leqq a_1 + W(A - \{a_1, a_2\}).$$

Now, by our induction hypothesis:

$$W(A) \leqq a_1 + \frac{1}{3}\sum_{i=3}^{n} a_i \log\left(\frac{1}{a_i}\right) + \sum_{i=3}^{n} a_i$$

$$\leqq \frac{1}{3}\sum_{i=1}^{n} a_i \log\left(\frac{1}{a_i}\right) + \sum_{i=1}^{n} a_i.$$

Case 2. a_1 is an interior stick of $L(\mathbf{x})$. In this case we may assume that a_2 and a_3 are the neighbours of a_1 in $L(\mathbf{x})$ and that $a_2 \leqq a_3$.

Now, $W(A) = G(\mathbf{x}) \leqq a_1 + W(A - \{a_1, a_2, a_3\} \cup \{a_1 + a_2 + a_3\})$. By our induction hypothesis,

$$W(A) \leqq a_1 + \frac{1}{3}\sum_{i=4}^{n} a_i \log\left(\frac{1}{a_i}\right) + \frac{1}{3}(a_1 + a_2 + a_3) \log\left(\frac{1}{a_1 + a_2 + a_3}\right)$$

$$+ \sum_{i=4}^{n} a_i + (a_1 + a_2 + a_3)$$

$$= \frac{1}{3}\sum_{i=1}^{n} a_i \log\left(\frac{1}{a_i}\right) + \sum_{i=1}^{n} a_i + a_1 + \frac{1}{3}(a_1 + a_2 + a_3) \log\left(\frac{1}{a_1 + a_2 + a_3}\right)$$

$$- \frac{1}{3} a_1 \log\left(\frac{1}{a_1}\right) - \frac{1}{3} a_2 \log\left(\frac{1}{a_2}\right) - \frac{1}{3} a_3 \log\left(\frac{1}{a_3}\right).$$

Fixing a_1 and $a_1 + a_2 + a_3$, since $a_1 \leqq a_2 \leqq a_3$, we know that $-a_2 \log(1/a_2) - a_3 \log(1/a_3)$ is maximized when $a_2 = a_1$. Furthermore, fixing a_1 and fixing $a_1 = a_2$ we see that

$$\frac{1}{3}(a_1 + a_2 + a_3) \log\left(\frac{1}{a_1 + a_2 + a_3}\right) - \frac{1}{3} a_3 \log\left(\frac{1}{a_3}\right)$$

is maximized when $a_3 = a_1$. Thus,

$$W(A) \leqq \frac{1}{3}\sum_{i=1}^{n} a_i \log\left(\frac{1}{a_i}\right) + \sum_{i=1}^{n} a_i + a_1 + \frac{1}{3}(3a_1) \log\left(\frac{1}{3a_1}\right) - 3\left(\frac{1}{3} a_1 \log\left(\frac{1}{a_1}\right)\right)$$

$$= \frac{1}{3}\sum_{i=1}^{n} a_i \log\left(\frac{1}{a_i}\right) + \sum_{i=1}^{n} a_i. \qquad \square$$

Now, for any bag $A = \{a_1, \cdots, a_{n-1}\}$ of sticks,

$$\frac{1}{3}\sum_{i=1}^{n-1} a_i \log\left(\frac{1}{a_i}\right) \leqq \frac{1}{3}\log(n-1) \quad \text{and} \quad \sum_{i=1}^{n-1} a_i \leqq 1.$$

Thus, from our proposition, the greedy matching applied to a set of n points constructs a matching of length at most $\frac{1}{3}\log(n-1) + 1$. We note that this implies our examples are the worst possible for n of the form $3^k + 1$. Indeed, it is easy to see from our proposition that they are unique such examples.

Acknowledgments. Our thanks to Mike Saks and Bill Steiger for helpful discussions.

REFERENCES

[1] D. AVIS, *Worst case bounds for the Euclidean matching problem*, Comput. Math. Appl., 7 (1981), pp. 251-257.

[2] D. AVIS, B. DAVIS, AND M. STEELE, *Probabilistic analysis of a greedy heuristic for Euclidean matching*, Probab. in the Eng. Inform. Sci., 2 (1988), pp. 143–156.

[3] W. FELLER, *An Introduction to Probability Theory and Its Applications, Vol. II*, Second edition, John Wiley, New York, 1971.

[4] E. M. RHEINGOLD AND R. E. TARJAN, *On a greedy heuristic for complete matching*, SIAM J. Comput., 10 (1981), pp. 676–681.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 673-677, August 1990

007

A NOTE ON BENNETT'S TIME–SPACE TRADEOFF FOR REVERSIBLE COMPUTATION*

ROBERT Y. LEVINE† AND ALAN T. SHERMAN‡

Abstract. Given any irreversible program with running time T and space complexity S, and given any $\varepsilon > 0$, Bennett shows how to construct an equivalent reversible program with running time $O(T^{1+\varepsilon})$ and space complexity $O(S \ln T)$. Although these loose upper bounds are formally correct, they are misleading due to a hidden constant factor in the space bound. It is shown that this constant factor is approximately $\varepsilon 2^{1/\varepsilon}$, which diverges exponentially as ε approaches 0. Bennett's analysis is simplified using recurrence equations and it is proven that the reversible program actually runs in time $\Theta(T^{1+\varepsilon}/S^{\varepsilon})$ and space $\Theta(S(1+\ln(T/S)))$.

Bennett claims that for any $\varepsilon > 0$, the reversible program can be made to run in time $O(T)$ and space $O(ST^{\varepsilon})$. This claim is corrected and tightened as follows: whenever $T \geqq 2S$ and for any $\varepsilon \geqq 1/(0.58 \lg(T/S))$, the reversible program can be made to run in time $\Theta(T)$ and space $\Omega(S(T/S)^{\varepsilon/2}) \cap O(S(T/S)^{\varepsilon})$. For $S \leqq T < 2S$, Bennett's 1973 simulation yields an equivalent reversible program that runs in time $\Theta(T)$ and space $\Theta(S)$.

Key words. algorithms, reversible computation, time-space tradeoff

AMS(MOS) subject classifications. 68Q05, 68Q15

1. Introduction.

A Turing machine is *reversible* if and only if its state-transition function is injective. In other words, a program is reversible if for any input and for any state in the program execution on that input, the preceding state is uniquely determined from the current state. For example, the program "On input x and y, output $x+y$." is not reversible because the input cannot be determined from the output, but the related program "On input x and y, output $(x+y, x)$." is reversible. The notion of reversible computation might someday radically alter the design of computers because there are models of reversible computation in which computations do not dissipate heat [3].

In 1973 Bennett [2] presented a general method for transforming any irreversible program into an equivalent reversible program. This method works by reversibly simulating the irreversible program. Let T and S denote, respectively, the time and space bounds of the irreversible program. Bennett proved that his simulation runs in time $\Theta(T)$ and space $\Theta(T+S)$. Thus, because T can be exponential in S, the simulation takes space exponential in S in the worst case.

In 1988 Bennett [1] improved his simulation by introducing a time-space tradeoff. He proved the following result: given any $\varepsilon > 0$, the revised simulation can be made to run in time $O(T^{1+\varepsilon})$ and space $O(S \ln T)$.[1] Although these loose upper bounds are formally correct, they are misleading because there is a large hidden constant factor in the space bound that depends on ε; in fact, the tradeoff is between the exponent in the reversible time and the constant factor in the reversible space. In this note we exactly compute the constant factor in the space bound and show that it is approximately

* Received by the editors May 15, 1989; accepted for publication (in revised form) October 5, 1989.

† Massachusetts Institute of Technology, Lincoln Laboratory, Lexington, Massachusetts 02173-0073. This work was done while this author was a student at Tufts University.

‡ Computer Science Department, University of Maryland, Baltimore, Maryland 21228, and Institute for Advanced Computer Studies, University of Maryland, College Park, Maryland 20742.

[1] Throughout this paper let $\ln = \log_e$ and $\lg = \log_2$.

$\varepsilon 2^{1/\varepsilon}$. This constant factor diverges as ε approaches 0, which is the interesting case in which the reversible time approaches the irreversible time. Using recurrence equations we also prove that Bennett's revised simulation actually runs in time $\Theta(T^{1+\varepsilon}/S^{\varepsilon})$ and space $\Theta(S(1+\ln(T/S)))$.

Bennett achieves his improved simulation by embedding his 1973 technique in a clever general-purpose time-space tradeoff. Although unknown to Bennett, this general-purpose time-space tradeoff was independently discovered by Chandra [5] in 1972.[2]

The rest of this note is divided into two sections. In § 2 we give a simplified analysis of Bennett's time-space tradeoff using recurrence equations. We also plot the tradeoff curve for the reversible time and space for several values of T/S. In § 3 we compute the constant factor in the space bound and correct an error in Bennett's paper.

2. Time and space analysis using recurrence equations. Consider any irreversible program that runs in time T and space S. We assume that $T \geqq 2S$, since otherwise it would be better to use Bennett's 1973 algorithm to produce an equivalent reversible program that runs in time $\Theta(T)$ and space $\Theta(S)$. Bennett's 1988 tradeoff algorithm depends on three integral parameters $m \geqq 1$, $k \geqq 2$, and $n \geqq 0$. The tradeoff algorithm reversibly simulates the irreversible program in segments of m steps using Bennett's 1973 algorithm. For $n > 0$, a total of k^n m-step segments are simulated by performing k forward and $k-1$ backward simulations each consisting of mk^{n-1} steps. For $n = 0$, the 1973 algorithm is used. At the end of each forward simulation only the final configuration is stored. At the end of each backward simulation one previously stored configuration is erased. Using Bennett's notation, let $R(z, x, n, m, d)$ represent the reversible simulation of mk^n steps of the original irreversible program from configuration z to configuration x in segments of size m. The parameter d, which takes on the values 1 and -1, signals whether a forward or backward simulation is taking place.

We now consider the associated recurrence equations for the time and space complexity of the simulation. Let P_n and Q_n be the number of steps in $R(z, x, n, m, d)$ with $d = 1$ and with $d = -1$, respectively. We then have the following coupled recurrence equations

$$P_n = \begin{cases} kP_{n-1} + (k-1)Q_{n-1} & \text{if } n > 0 \\ m & \text{if } n = 0 \end{cases} \tag{1}$$

and

$$Q_n = \begin{cases} kQ_{n-1} + (k-1)P_{n-1} & \text{if } n > 0 \\ m & \text{if } n = 0. \end{cases} \tag{2}$$

Substituting (1) into (2) yields the second-order recurrence

$$P_n = \begin{cases} 2kP_{n-1} - (2k-1)P_{n-2} & \text{if } n > 1 \\ m(2k-1) & \text{if } n = 1 \\ m & \text{if } n = 0, \end{cases} \tag{3}$$

which can be solved exactly by characteristic equations [4] to obtain

$$P_n = m(2k-1)^n. \tag{4}$$

The space bound S_n satisfies the recurrence equation

$$S_n = \begin{cases} (k-1)m + S_{n-1} & \text{if } n > 1 \\ (k-1)m & \text{if } n = 1, \end{cases} \tag{5}$$

[2] For another interesting application of this tradeoff, see the "cycling known-plaintext attack" against group ciphers by Kaliski, Rivest, and Sherman [7].

which describes the number of stored intermediate configurations. By iteration [6] the exact solution is

$$S_n = mn(k-1). \tag{6}$$

Bennett chooses $m = S$ to ensure that each stored configuration takes at most S space. From this assumption (4) and (6) yield, respectively, the following time bound T' and space bound S' for the reversible program

$$T' = S(2k-1)^n \tag{7}$$

and

$$S' = Sn(k-1). \tag{8}$$

Assuming

$$T = Sk^n, \tag{9}$$

that is, assuming the number of simulated steps Sk^n is equal to the running time of the irreversible program, we have $n = (\ln(T/S))/\ln k$. Thus by appropriate choice of k, the user can select any point along the tradeoff curve

$$\frac{T'}{S'} = \frac{(2k-1)^n}{n(k-1)}. \tag{10}$$

Figure 1 shows the logarithm of this tradeoff curve for several values of T/S.

We now separately express T' and S' in terms of T, S, and k. By (7) and (9),

$$\frac{T'}{T} = \left(\frac{2k-1}{k}\right)^n = (2-(1/k))^{(\ln(T/S))/\ln k} = \left(\frac{T}{S}\right)^{\epsilon(k)} \tag{11}$$

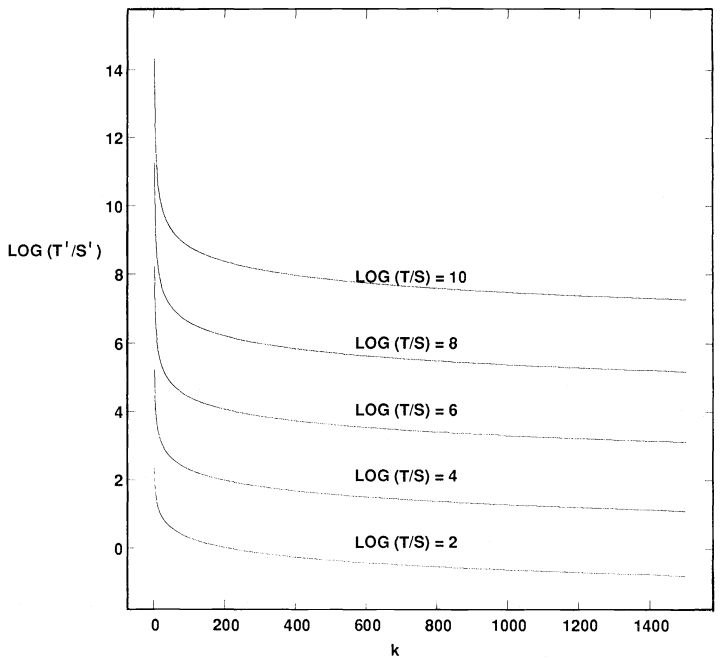


FIG. 1. *Tradeoff curves* $\log_{10}(T'/S')$ *for the reversible time and space as a function of the parameter* k *and the ratio* T/S *of the irreversible time and space. The curves are drawn for* $T/S = 10^2, 10^4, 10^6, 10^8, 10^{10}$.

and thus

$$T' = T\left(\frac{T}{S}\right)^{\varepsilon(k)} = \frac{T^{1+\varepsilon(k)}}{S^{\varepsilon(k)}} \tag{12}$$

where

$$\varepsilon(k) = \frac{\ln(2-(1/k))}{\ln k}. \tag{13}$$

Similarly by (8),

$$S' = S\frac{\ln(T/S)}{\ln k}(k-1) = c(k)S\ln\frac{T}{S} \tag{14}$$

where

$$c(k) = \frac{k-1}{\ln k}. \tag{15}$$

Therefore, regardless of the relationship between T and S, given any $\varepsilon > 0$ the reversible program can be made to run in time $\Theta(T^{1+\varepsilon}/S^{\varepsilon})$ and space $\Theta(S(1+\ln(T/S)))$. Equations (12) and (14) differ from Bennett's corresponding bounds in two respects. First, our bounds are tight and include the dependence of T' and S' on S.[3] Second, the ε used by Bennett is equal to $1/\lg k$, which is slightly larger and approximately equal to $\varepsilon(k)$ for large k.

3. Calculation of the constant factor in the space bound. The constant factor for the space bound is the term $c(k)$ in (14). To express this factor in terms of $\varepsilon(k)$ we first compute the Taylor series expansion of $\varepsilon(k)$ around $1/k$ to obtain

$$\varepsilon(k) = \frac{1}{\ln k}\left(\ln 2 - \frac{1}{2k} - \frac{1}{8k^2} - \cdots\right). \tag{16}$$

Letting

$$\varepsilon_0(k) = \frac{\ln 2}{\ln k} = \frac{1}{\lg k} \tag{17}$$

be the first term in this expansion (i.e., $\varepsilon_0(k)$ is Bennett's ε), we have $\ln k = (\ln 2)/\varepsilon_0(k)$ and $k = 2^{1/\varepsilon_0(k)}$. Therefore, by (15)

$$c(k) = \frac{\varepsilon_0(k)}{\ln 2}(2^{1/\varepsilon_0(k)} - 1) \sim \frac{\varepsilon_0(k)}{\ln 2}2^{1/\varepsilon_0(k)}. \tag{18}$$

The most interesting case is when $\varepsilon(k)$ tends to 0—that is, when the reversible time approaches the irreversible time—but in this case the term $c(k)$ diverges.

To illustrate how the constant factor $c(k)$ diverges when $\varepsilon(k)$ approaches 0, consider what happens when $aT \leqq T' \leqq bT$ for some constants $1 < a \leqq b$. In this case, the identity $\varepsilon(k) = (\ln(T'/T))/\ln(T/S)$ from (12) implies that

$$\frac{\ln a}{\ln(T/S)} \leqq \varepsilon(k) \leqq \frac{\ln b}{\ln(T/S)}. \tag{19}$$

Note that to be able to find an integer k that satisfies (19), it is necessary for

[3] Although Bennett did not explicitly state the dependence of T' and S' on S, the details of his proof given in the appendix of [1] suggest that he was aware of this dependence.

$a \leqq (T/S)^{\varepsilon_{\max}}$, where $\varepsilon_{\max} = \max\{\varepsilon(k): k \geqq 2\} = \varepsilon(2) \approx 0.58$. Since $k \geqq 2$, (18) implies that

$$\frac{\varepsilon_0(k)}{2 \ln 2} 2^{1/\varepsilon_0(k)} \leqq c(k) < \frac{\varepsilon_0(k)}{\ln 2} 2^{1/\varepsilon_0(k)}. \tag{20}$$

Using the fact that $\varepsilon(k) < \varepsilon_0(k) < 2\varepsilon(k)$, it follows that

$$\frac{\varepsilon(k)}{2 \ln 2} 2^{1/(2\varepsilon(k))} < c(k) < \frac{2\varepsilon(k)}{\ln 2} 2^{1/\varepsilon(k)}. \tag{21}$$

By (19),

$$\frac{(\ln a)/\ln(T/S)}{2 \ln 2} 2^{(\ln(T/S))/(2 \ln b)} < c(k) < \frac{(2 \ln b)/\ln(T/S)}{\ln 2} 2^{(\ln(T/S))/\ln a} \tag{22}$$

and hence

$$\frac{\lg a}{2 \ln(T/S)} \left(\frac{T}{S}\right)^{1/(2 \lg b)} < c(k) < \frac{2 \lg b}{\ln(T/S)} \left(\frac{T}{S}\right)^{1/\lg a}. \tag{23}$$

Equation (14) then yields the space bounds

$$\frac{\lg a}{2} S \left(\frac{T}{S}\right)^{1/(2 \lg b)} < S' < 2(\lg b) S \left(\frac{T}{S}\right)^{1/\lg a}, \tag{24}$$

which grow superlogarithmically in T/S.

We conclude with one more refinement of Bennett's results. Bennett claims that for any $\delta > 0$, the reversible program can be made to run in time $O(T)$ and space $O(ST^{\delta})$. But as Bennett notes, his δ depends on T; hence, Bennett's argument does not hold for arbitrary δ. We correct Bennett's claim as follows. For any $\delta \geqq 1/(\varepsilon_{\max} \lg(T/S))$, we can choose k such that $a = b = 2^{1/\delta}$, in which case (24) yields the tighter space bound

$$\frac{1}{2\delta} S \left(\frac{T}{S}\right)^{\delta/2} < S' < \frac{2}{\delta} S \left(\frac{T}{S}\right)^{\delta}. \tag{25}$$

Equation (25) proves that whenever $T \geqq 2S$ and for any $\delta \geqq 1/(\varepsilon_{\max} \lg(T/S))$, the reversible program can be made to run in time $\Theta(T)$ and space $\Omega(S(T/S)^{\delta/2}) \cap O(S(T/S)^{\delta})$.

Although Bennett's clever time-space tradeoff provides a better way to transform any irreversible program into an equivalent reversible program, its practical utility is diminished by its huge constant factor in the space bound. It remains an open question whether or not Bennett's algorithm yields an optimum tradeoff.

REFERENCES

[1] C. H. BENNETT, *Time/space trade-offs for reversible computation*, SIAM J. Comput., 18 (1989), pp. 766–776.

[2] ———, *Logical reversibility of computation*, IBM J. Res. Devel., 17 (1973), pp. 525–532.

[3] C. H. BENNETT AND R. LANDAUER, *The fundamental physical limits of computation*, Scientific American, 253 (1985), pp. 48–56.

[4] G. BRASSARD AND P. BRATLEY, *Algorithms: Theory & Practice*, Prentice-Hall, Englewood Cliffs, NJ, 1988.

[5] A. K. CHANDRA, *Efficient compilation of linear recursive programs*, Stanford Artificial Intelligence Project Memo AIM-167, STAN-CS-72-282, Computer Science Department, Stanford University, Stanford, CA, April 1972.

[6] T. CORMEN, C. E. LEISERSON, AND R. L. RIVEST, *Introduction to Algorithms*, MIT Press and McGraw-Hill, Cambridge, MA, 1990, to appear.

[7] B. S. KALISKI JR., R. L. RIVEST, AND A. T. SHERMAN, *Is the data encryption standard a group? (Results of cycling experiments on* DES), J. Cryptology, 1 (1988), pp. 3–36.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 678–704, August 1990

008

PLANAR DEPTH-FIRST SEARCH IN $O(\log n)$ PARALLEL TIME*

TORBEN HAGERUP†

Abstract. It is shown that, in the PRIORITY CRCW PRAM model and for the class of undirected embedded planar graphs, depth-first search (DFS) trees are no more difficult to construct than breadth-first search (BFS) trees.

Specifically, suppose that time $T(n) \geqslant \log n$ and $p(n)$ processors suffice to construct a planar embedding of a planar graph on n vertices and to compute a BFS tree of an undirected connected planar graph on $3n$ vertices. Then, given an undirected connected planar graph G on n vertices, a DFS tree of G can be computed in the stated model in $O(T(n))$ time with $p(n)$ processors.

By using known results for the above problems, a DFS tree construction algorithm that runs in $O(\log n)$ time and uses $O(n^3)$ processors is derived. The fastest previously known algorithm has time and processor bounds of $O((\log n)^2)$ and $O(n)$, respectively.

Key words. depth-first search, planar graphs, parallel computing

AMS(MOS) subject classifications. 68C05, 68C25, 68E10

1. Introduction.

Interest in the problem of parallel construction of DFS trees in directed or undirected graphs (the directed or undirected *DFS tree problem*) stems from two sources. On the one hand, depth-first search has proved to be an extremely valuable technique in sequential graph algorithms [18], and a number of sequential algorithms could easily be turned into interesting parallel algorithms if DFS trees could be made available at a sufficiently low cost. On the other hand, depth-first search has been conjectured by many to be hard or impossible to parallelize—"inherently sequential," to use a popular buzz word [17]. Hence we can also be interested in the DFS tree problem from a more complexity-oriented point of view. Reif [15] has shown the problem to be P-complete if we insist on obtaining the DFS tree constructed by a certain simple sequential algorithm. However, this seems a severe and artificial restriction in the context of parallel algorithms, and indeed some positive results have been obtained recently. Aggarwal and Anderson [1] and Aggarwal, Anderson, and Kao [2] describe RNC algorithms to compute DFS trees in undirected and directed graphs, respectively (see [4] for definitions of the complexity classes NC and RNC), and Goldberg, Plotkin, and Vaidya [9] and Aggarwal, Anderson, and Kao [2] have given deterministic PRAM algorithms for the undirected and directed DFS tree problem, respectively, that use $O(n^3)$ processors and $O(\sqrt{n}(\log n)^{O(1)})$ time on n-vertex input graphs.

At the time of writing, neither the directed nor the undirected DFS tree problem is known to belong to NC. Hence both motivations mentioned above lead naturally to the consideration of more tractable special cases of the DFS tree problem. This paper considers one such special case, that of undirected planar graphs. Smith [17] showed the DFS tree problem for such graphs to be in NC by giving an algorithm that runs in $O((\log n)^3)$ time on a CREW PRAM with $O(n^4)$ processors. His method is based on the divide-and-conquer strategy, whereby problem instances are divided by means of a *cyclic separator*, a simple cycle in the graph with the property that neither

* Received by the editors December 1, 1987; accepted for publication (in revised form) October 10, 1989. This research was supported by the Deutsche Forschungsgemeinschaft, SFB 124, TP B2, VLSI Entwurfsmethoden und Parallelität.

† Fachbereich Informatik, Universität des Saarlandes, D-6600 Saarbrücken, Federal Republic of Germany.

its inside nor its outside contains more than $\frac{2}{3}$ of the graph's vertices. The same idea was used in subsequent improvements culminating in algorithms by Goldberg, Plotkin, and Shannon [8], [16] that use $O((\log n)^2)$ time and $O(n)$ processors on a CRCW PRAM, or $O((\log n)^2 \log^* n)$ time and $O(n)$ processors on a CREW PRAM. A recent NC algorithm by Kao [13] for the directed DFS tree problem in planar graphs also recursively computes cyclic separators (defined somewhat differently). Since the depth of recursion can be $\Theta(\log n)$ and the construction of a cyclic separator is a nontrivial problem, a different approach seems to be needed if we want to achieve a logarithmic running time.

We give an algorithm for the undirected DFS tree problem in planar graphs based on a direct analysis of the structure of a planar graph imposed by a BFS tree of its face incidence graph. It runs in time $O(\log n)$ and is the fastest algorithm known. The number of processors is high, $\Theta(n^3)$ on the PRIORITY CRCW PRAM. However, we show that, except for the construction of a planar embedding and of a BFS tree of a planar graph, all steps of the algorithm can be executed optimally with $O(n/\log n)$ processors. Hence, while there may be little hope of overcoming the bottlenecks mentioned, they have been clearly identified, and we have reduced the DFS tree problem for undirected planar graphs to two maybe more central problems.

2. Preliminaries. This section introduces notation and terminology used in the remainder of the paper, most of which is standard, and mentions some elementary facts.

When f is a function and U is a subset of its domain, let $f(U) = \bigcup_{u \in U} f(u)$. For $k \geq 1$, $f^{(k)}$ denotes k-fold repeated application of f, i.e., $f^{(1)} = f$ and $f^{(i+1)} = f \circ f^{(i)}$, for $i \geq 1$. For $n \geq 1$, $\log^* n = \min \{i \geq 1 \mid \log_2^{(i)} n \leq 1\}$.

A *directed edge* over a set V is an element of $V \times V$. Given a directed edge $e = (u, v)$, u and v are called the *endpoints* of e; u is its *tail*, and v its *head*. A directed edge is said to *leave* its tail and to *enter* its head. An *undirected edge* over V is a subset of V containing exactly two elements. The endpoints of an undirected edge are its two elements. A directed or undirected edge is said to be *incident* on its endpoints. If $e = (u, v)$ is a directed edge with $u \neq v$, the *undirected version* of e is the undirected edge $\{u, v\}$. The *directed versions* of an undirected edge $\{u, v\}$ are the two directed edges (u, v) and (v, u).

Given a finite set $V \neq \emptyset$, a *directed* (*undirected*, respectively) *graph* on the vertex set V is a pair $G = (V, E)$, where E is a set of directed (undirected) edges over V. The elements of V are called the *vertices* and the elements of E the *edges* of G (sometimes, depending on context, we say "in G" or "on G" instead of "of G"). The elements of $V \cup E$ are collectively called the *elements* of G, and a graph is said to *contain* its elements. Given a graph $G = (V, E)$, $\mathcal{V}(G)$ denotes its vertex set V. We extend this notation to sets of graphs by defining $\mathcal{V}(\mathcal{G}) = \bigcup_{G \in \mathcal{G}} \mathcal{V}(G)$, for all sets $\mathcal{G}$ of graphs. Graphs $G_1, \cdots, G_l$ are (*vertex-*)*disjoint* if $\mathcal{V}(G_i) \cap \mathcal{V}(G_j) = \emptyset$, for $1 \leq i < j \leq l$.

Two vertices in a graph $G = (V, E)$ are called *adjacent* in G if they are the endpoints of a common edge in G, and a subset $V' \subseteq V$ is called an *independent* vertex set in G if no two vertices in V' are adjacent in G. The *indegree* (*outdegree*) of a vertex u in a directed graph G is the number of edges in G entering (leaving) u. The *degree* of a vertex u in an undirected graph G is the number of edges in G incident on u. Given a directed graph $G = (V, E)$, the *undirected version* of G is the undirected graph $(V, \{\{u, v\} \mid (u, v) \in E \text{ and } u \neq v\})$. Given an undirected graph $G = (V, E)$, the *directed version* of G is the directed graph $(V, \{(u, v) \mid \{u, v\} \in E\})$, i.e., each undirected edge is replaced by its two directed versions. The edges of the directed version of an undirected graph G are called the *darts* of G. Given two undirected graphs $G = (V, E)$ and

$G'=(V', E')$, an *isomorphism* from G to G' is a bijection $h: V \to V'$ such that for all $u, v \in V: (\{h(u), h(v)\} \in E' \Leftrightarrow \{u, v\} \in E)$, and G and G' are said to be *isomorphic* if there exists an isomorphism from G to G'.

A *subgraph* of a graph (V, E) is a graph (V', E') with $V' \subseteq V$ and $E' \subseteq E$. A graph is said to contain its subgraphs. The subgraph of a graph (V, E) *spanned* by a vertex set $V' \subseteq V$ is the graph (V', E'), where $E' \subseteq E$ consists of those elements of E that are edges over V'. The subgraph of (V, E) spanned by an edge set $E' \subseteq E$ is the graph (V', E'), where $V' \subseteq V$ is the set of endpoints of edges in E'. Given directed graphs $G_1=(V_1, E_1), \cdots, G_l=(V_l, E_l)$, their *sum*, denoted by $G_1 \cup \cdots \cup G_l$, is the directed graph $(V_1 \cup \cdots \cup V_l, E_1 \cup \cdots \cup E_l)$. A *bipartite* graph on disjoint vertex sets V and W is a graph G on the vertex set $V \cup W$ such that each edge in G has one endpoint in each of V and W. A *maximal* graph with a given property is a graph that has the property, but is not properly contained in any other graph with the property.

The (directed, simple) *path* $(u_0, \cdots, u_l)$, for $l \geqq 0$ and $u_i \neq u_j$ for $0 \leqq i < j \leqq l$, is the directed graph $(\{u_0, \cdots, u_l\}, \{(u_i, u_{i+1}) \mid 0 \leqq i < l\})$. The vertices on the path are said to occur in the order $u_0, \cdots, u_l$, its edges in the order $(u_0, u_1), \cdots, (u_{l-1}, u_l)$, and the path is said to be from u_0 to u_l. l is called the *length* of the path. The *internal vertices* on the path are $u_1, \cdots, u_{l-1}$. A path in a directed graph G is a subgraph of G which is a path. A path in an undirected graph G is a path in the directed version of G. Given two vertices u and v in a graph G, the *distance* from u to v in G is the minimal length of a path in G from u to v, or ∞ if no such path exists.

The directed *simple cycle* $(u_0, \cdots, u_{l-1})$, for $l \geqq 1$ and $u_i \neq u_j$ for $0 \leqq i < j < l$, is the directed graph $(\{u_0, \cdots, u_{l-1}\}, \{(u_i, u_{(i+1) \bmod l}) \mid 0 \leqq i < l\})$. A simple cycle in a directed graph G is a subgraph of G that is a directed simple cycle. A simple cycle in an undirected graph G is the undirected version of a simple cycle with at least three vertices in the directed version of G. A graph is *acyclic* if it contains no simple cycles. For $i=0, \cdots, l-1$, $u_{(i+1) \bmod l}$ is called the *successor* and $u_{(i-1) \bmod l}$ the *predecessor* of u_i on the directed simple cycle $(u_0, \cdots, u_{l-1})$. The *nearest successor* with a given property of a vertex u on a directed simple cycle $C=(V, E)$ is the first vertex in the sequence $suc(u), suc^{(2)}(u), \cdots$ that has the property, where $suc(v)$, for $v \in V$, denotes the successor of v on C. The *nearest predecessor* is defined analogously.

A *segment* of a directed simple cycle $C=(V, E)$ is a subset $I \subseteq V$ that is either empty or is the vertex set of a path in C. I is called *trivial* if $I=\emptyset$ or $I=V$. Given a nontrivial segment I of a directed simple cycle C, we denote by $First_C(I)$ and $Last_C(I)$ the first and the last vertex, respectively, on the unique path p in C with $\mathcal{V}(p)=I$. We define a *cyclic order* of a finite set $V \neq \emptyset$ to be a directed simple cycle on the vertex set V. Given a directed simple cycle $C=(V, E)$ and a nonempty subset $V' \subseteq V$, the *cyclic order of V' on C* is defined to be the cyclic order $(u_0, \cdots, u_{l-1})$, where $V'=\{u_0, \cdots, u_{l-1}\}$ and $u_{(i+1) \bmod l}$ is the nearest successor of u_i on C belonging to V', for $i=0, \cdots, l-1$.

Given a vertex u in a graph $G=(V, E)$, a vertex $v \in V$ is *reachable* from u in G exactly if there is a path in G from u to v. Given an undirected graph $G=(V, E)$, we may define an equivalence relation on V by letting two vertices in G be equivalent exactly if one is reachable from the other in G. The subgraphs of G spanned by the equivalence classes of this relation are called the *connected components* of G. If G has only one connected component, G is *connected*.

Given an undirected graph $G=(V, E)$, we may define an equivalence relation on E by declaring two edges e_1 and e_2 in G equivalent exactly if $e_1=e_2$, or if there is a simple cycle in G containing both e_1 and e_2. The (not necessarily disjoint) subgraphs of G spanned by the equivalence classes of this relation are called the biconnected

components or *blocks* of G. An *articulation point* of G is a vertex in G that belongs to more than one of G's blocks. If some block of G contains just one edge, that edge is called a *bridge* in G. If G is connected and has only one block, G is called *biconnected.* Given a connected undirected graph G, a vertex r in G and a block B of G, there is a vertex in B whose distance from r in G is strictly less than that of any other vertex in B. We call this vertex the *r-dominator* of B in G.

A *directed tree* is an acyclic directed graph $T=(V, E)$ in which all vertices have outdegree one, except that precisely one vertex r has outdegree zero. r is called the *root* of T, and T is said to be rooted at r. For all $u \in V\backslash\{r\}$, the unique edge in T leaving u is called u's *parent pointer* in T, and the head of u's parent pointer is u's *parent* in T. For all $u, v \in V$, u is a *child* of v in T exactly if v is the parent of u in T, and u is a *descendant* of v in T and v an *ancestor* of u in T exactly if v is reachable from u in T. For all $u \in V$, the *depth* of u in T is the distance in T from u to r. A *directed forest* is a sum of vertex-disjoint directed trees. When talking about the trees in a forest, we will always mean its maximal trees. An *undirected tree* is a connected acyclic undirected graph. Given a connected undirected graph $G=(V, E)$, an undirected (directed) *spanning tree* of G is an undirected (directed) tree on the vertex set V that is a subgraph of G (of the directed version of G). Given a directed tree $T=(V, E)$, an undirected edge $\{u, v\}$ over V is called a *cross edge* relative to T if T contains neither a path from u to v nor a path from v to u. Given an undirected connected graph $G=(V, E)$, a DFS *tree* of G is a directed spanning tree T of G such that G contains no cross edges relative to T. A BFS *tree* of G is a directed spanning tree T of G such that for all $u \in V$, the depth of u in T is equal to the distance in G from u to the root of T. The definitions of DFS and BFS trees are motivated by the concepts known as depth-first and breadth-first search. For instance, a directed spanning tree T of G is a DFS tree of G exactly if there is a depth-first search of G such that an edge $(u, v) \in V \times V$ belongs to T if and only if u is discovered during the search via the edge $\{u, v\}$. Since we make no use of this connection except by way of motivation and in informal comments, we will not elaborate on this point but refer the reader, in the case of DFS trees, to the treatment in [18], in particular, to its Theorem 1.

A (*topological*) *planar embedding* of an undirected graph $G=(V, E)$ is a function $\mathscr{E}$ that maps the vertices of G to distinct points in $\mathbb{R}^2$ and each edge $\{u, v\} \in E$ to a Jordan curve in $\mathbb{R}^2$ from $\mathscr{E}(u)$ to $\mathscr{E}(v)$ such that for all $e=\{u, v\} \in E$, $\mathscr{E}(e) \cap (\mathscr{E}(V) \cup \mathscr{E}(E\backslash\{e\})) = \{\mathscr{E}(u), \mathscr{E}(v)\}$ (i.e., edges do not cross). G is *planar* if there exists a planar embedding of G.

Let $\mathscr{E}$ be a planar embedding of a planar graph $G=(V, E)$. The *faces* of $\mathscr{E}$ are the connected regions of $\mathbb{R}^2\backslash\mathscr{E}(V \cup E)$. The *boundary* (induced by $\mathscr{E}$) of a face F of $\mathscr{E}$ is the subgraph of G consisting of those elements $\alpha \in V \cup E$ for which each point of $\mathscr{E}(\alpha)$ is arbitrarily close to points in F. If G is biconnected and $|V| \geqq 3$, the boundary of each face of $\mathscr{E}$ is a simple cycle. Let D be the set of darts of G, and for each dart $e=(u, v) \in D$, let $\Phi_{\mathscr{E}}(e)$ be the dart $e'=(u, w) \in D$ such that $\mathscr{E}(\{u, w\})$ is the first curve in $\mathscr{E}(E)$ with endpoint $\mathscr{E}(u)$ encountered after $\mathscr{E}(\{u, v\})$ in a clockwise scan around $\mathscr{E}(u)$. $\Phi_{\mathscr{E}}$ is a permutation of D known as the *combinatorial planar embedding* corresponding to $\mathscr{E}$. Whereas our proofs are based on properties of topological embeddings, our algorithms work with combinatorial embeddings. Hence, for instance, "Embed G in the plane" means "Compute $\Phi_{\mathscr{E}}$ for some topological planar embedding $\mathscr{E}$ of G." The graph $(D, \{((u, v), \Phi_{\mathscr{E}}((v, u))) | (u, v) \in D\})$, which is a sum of vertex-disjoint directed simple cycles, is called the *face cycle graph* of $\mathscr{E}$, and its cycles the *face cycles* of $\mathscr{E}$. The intuitive meaning of a face cycle C of $\mathscr{E}$ is that it corresponds to a walk inside a particular face F of $\mathscr{E}$ along the image R under $\mathscr{E}$ of the elements on the

boundary of F, always keeping F to the left and R to the right (see Fig. 1). The face cycle C is said to be a face cycle of F. If G is connected, each face of $\mathscr{E}$ has precisely one face cycle. A face is said to be incident on the vertices and edges on its boundary, as well as on the darts on its face cycles. For all $u \in V$, let $D_u \subseteq D$ be the set of darts in G leaving u. For all $u \in V$ with $D_u \neq \emptyset$, we will call $(D_u, \{(e, \Phi_{\mathscr{E}}(e)) \mid e \in D_u\})$ the *cyclic order around u* (induced by $\mathscr{E}$). We also use this term for the cyclic orders derived in the obvious way of the edges and, if G is biconnected, of the faces of $\mathscr{E}$ incident on u.

We define an *embedded graph* somewhat loosely as an undirected planar graph G together with a particular planar embedding of G. We consider both the attributes of G and the attributes of its planar embedding to be attributes of the embedded graph so that, e.g., we can refer to its faces as well as its vertices. Subgraphs of embedded graphs are considered to be embedded in the natural way.

Euler's formula [6] relates the number of vertices, edges, and faces of a connected embedded graph. If these three quantities are n, m, and f, respectively, then $n+f=m+2$. In particular, $m \leqq 3n$ for any planar graph.

Given an embedded graph G, the *face incidence graph* $G_{\mathscr{D}}$ of G is the undirected graph whose vertex set is the set $\mathscr{F}$ of faces of G, and that contains an edge $\{F_1, F_2\}$, for $F_1, F_2 \in \mathscr{F}$ and $F_1 \neq F_2$, exactly if the boundaries of F_1 and F_2 have at least one vertex in common. $G_{\mathscr{D}}$ clearly is connected.

An embedded simple cycle C has precisely two faces. Exactly one of these contains points that are arbitrarily far apart. It is called the *outside* of C, whereas the other face of C is called its *inside*. *Inside* C (*outside* C) means contained in the inside (outside) of C.

For many of the concepts defined above relative to a particular graph G, we will often leave the identity of G to be deduced from the context. We allow ourselves to omit "the set of" in phrases such as "a graph on (the set of) n vertices."

A PRAM (parallel RAM) is a machine consisting of a finite number p of processors (RAMs) operating synchronously on an infinite global memory consisting of cells numbered $0, 1, \cdots$. We assume that the processors are numbered $1, \cdots, p$ and that each processor is able to access its own number. All processors execute the same program. We use the unit-cost model in which each memory cell can hold integers of size polynomial in the size of the input and each processor is able to carry out usual arithmetic operations including sign test, addition, subtraction, and multiplication and integer division by powers of two, but not necessarily general multiplication and integer division, on such numbers in constant time.

Various types of PRAMs have been defined, differing in the conventions regarding concurrent reading and writing, i.e., attempts by several processors to access the same memory cell in the same step. CRCW (concurrent-read concurrent-write) PRAMs

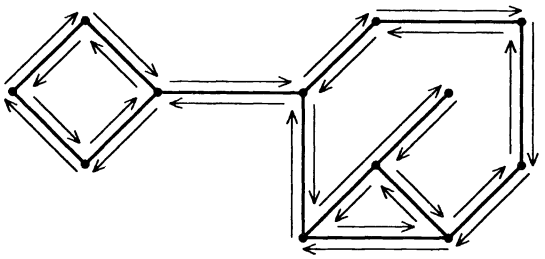


FIG. 1. *An embedded planar graph and its face cycles.*

allow simultaneous reading from as well as simultaneous writing to each cell by arbitrary sets of processors. Simultaneous writing is not immediately logically meaningful, and a further subclassification based on the write conflict resolution rule employed is standard. The algorithms in this paper use the PRIORITY (CRCW) PRAM, defined by the property that in the event of several processors attempting to write to the same memory cell in the same step, the lowest-numbered processor among them succeeds (i.e., the value that it attempts to write will actually be present in the cell after the write step). A CREW PRAM allows simultaneous reading from, but not simultaneous writing to a memory cell by several processors.

A parallel algorithm for a given problem using $p(n)$ processors and time $T(n)$ on inputs of size n is said to be *optimal* if its time-processor product $p(n)T(n)$ is at most a constant factor larger than the running time of the fastest known sequential algorithm to solve the problem. Whenever employed in this paper, the term is used in connection with problems that can be solved in linear sequential time. Hence "optimal" means "in $O(n/p(n))$ time and with $p(n)$ processors," for some $p(n)$.

Suppose that some PRAM computation can be carried out with $p(n)$ processors in time $T(n)$. Then, for any function $g(n) \geqq 1$ that is sufficiently easily computable (this will never be an issue in this paper), it can also be carried out with $O(p(n)/g(n))$ processors in time $O(g(n)T(n))$ (i.e., PRAM computations can always be slowed down). Note that this is true even for the PRIORITY PRAM model.

3. An informal description. Before we turn to the formal development of the algorithm, we provide a sketch of the basic ideas.

If we take as our starting point the idea of cyclic separators used in previous algorithms combined with a desire to reduce the running time below $\Theta((\log n)^2)$, a natural question to ask is whether it might be possible to compute all cyclic separators ever needed simultaneously, rather than in $\Theta(\log n)$ successive stages. Consider the following idealized picture of a planar graph (Fig. 2): There is one central face, and the other faces form concentric rings around the central face. For a graph with this

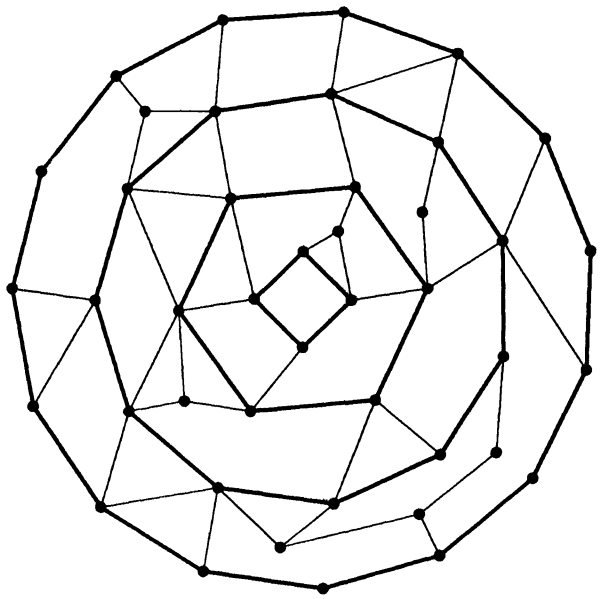


FIG. 2. *An idealized planar graph. Heavier lines denote black edges.*

structure, it is easy to identify a collection of disjoint simple cycles, namely, those that separate the rings, i.e., the cycles formed by those edges that separate faces at different distances from the central face in the face incidence graph. We call such edges *black edges*, and the cycles that they form *black cycles*. Although, due to the lack of control over the distribution of vertices, we cannot construe the black cycles as a collection of successive cyclic separators, they are eminently useful in the construction of a DFS tree. Suppose that we start a depth-first search at some vertex on the boundary of the central face, that we arbitrarily assign an orientation to each black cycle, and that we require the search to explore black edges first whenever there is a choice, and black edges only in the direction defined by the orientation of the black cycles (recall that the task is to compute *any* DFS tree). Suppose further for a moment that we know for each black cycle the first vertex on the cycle to be discovered by the search, which we call the *leader* of the cycle. Then the restriction of the corresponding DFS tree to each black cycle is completely fixed: It is a simple (directed) path containing all vertices on the cycle and ending in its leader. All that remains then is to compute the restriction of the DFS tree to the subgraphs that connect consecutive black cycles. Since these are treelike (which is one of the things that we have to make precise and to prove), this is not too difficult. Between two consecutive black cycles, exactly one tree near the leaders of the two cycles is explored in the direction from the inner to the outer cycle, whereas all other trees are explored in the opposite direction.

If the leader of a particular black cycle is known, it is an easy matter to determine the leader of the next enclosing black cycle. Hence by using a pointer-doubling technique, we can hope to spread information about leaders from the innermost to the outermost black cycle in logarithmic time. Indeed, a brute-force method solves this problem by reducing it to finding the transitive closure of a suitable matrix.

Not all planar graphs are of the simple form shown in Fig. 2. The black edges do not necessarily form a collection of disjoint black cycles, and even if they do, it may not be possible to order the black cycles such that each cycle encloses all the previous ones. The general situation, however, is not significantly more complicated, and the ideas developed above essentially carry through.

Section 4 derives those properties of planar graphs on which our algorithms are based, some of which were hinted at above. Section 5 presents a first and comparatively simple DFS tree construction algorithm that uses $\Theta(n^3)$ processors, whereas § 6 describes the modifications necessary to achieve optimality in all steps except the computation of a planar embedding and of a BFS tree.

4. Properties of embedded planar graphs. Throughout this section, let $G=(V, E)$ be an embedded undirected biconnected planar graph on at least three vertices, and let $r \in V$. Let $\mathcal{F}$ be the set of G's faces, D the set of its darts, and $G_{\mathcal{D}}$ its face incidence graph. When nothing else is stated, "face" means "element of $\mathcal{F}$." Let F_0 be an arbitrary face incident on r, called the *initial face*, and for all $F \in \mathcal{F}$, let $Type(F)$ be the distance in $G_{\mathcal{D}}$ from F_0 to F. Extend the notation to $V \cup E \cup D$ by defining

$$Type(\alpha) = \{k \geqq 0 \mid \alpha \text{ has an incident face } F \text{ with } Type(F) = k\},$$

for all $\alpha \in V \cup E \cup D$. For $\alpha \in V \cup E \cup D \cup \mathcal{F}$, $Type(\alpha)$ is called the *type* of α.

LEMMA 1. *For all $\alpha \in V \cup E$, $Type(\alpha)$ is an element of the sequence*

$$\{0, 1\}, \{1\}, \{1, 2\}, \{2\}, \{2, 3\}, \cdots.$$

If $e=\{u, v\} \in E$, then $Type(e) \subseteq Type(u)$, and $Type(e)$ and $Type(u)$ are either identical or adjacent elements in the sequence.

Proof. The fact is obvious. □

DEFINITION. For all $\alpha \in V \cup E$, α is called *white* if $|Type(\alpha)| = 1$, and *black* if $|Type(\alpha)| = 2$.

Let $G_{\mathcal{B}}$ be the subgraph of G spanned by the set of black edges. By Lemma 1, each endpoint of a black edge e is black and of the same type as e. Hence all vertices and edges within a connected component of $G_{\mathcal{B}}$ are of the same type.

Fig. 3 illustrates the concepts introduced so far.

Motivated by the following lemma, define a total order $<$ on the set of vertex and edge types by

$$\{0, 1\} < \{1\} < \{1, 2\} < \{2\} < \{2, 3\} < \cdots.$$

LEMMA 2. *If both the inside and the outside of a simple cycle C in G contains a face of type k, for some $k \geqq 0$, then C contains a vertex u with $Type(u) < \{k\}$.*

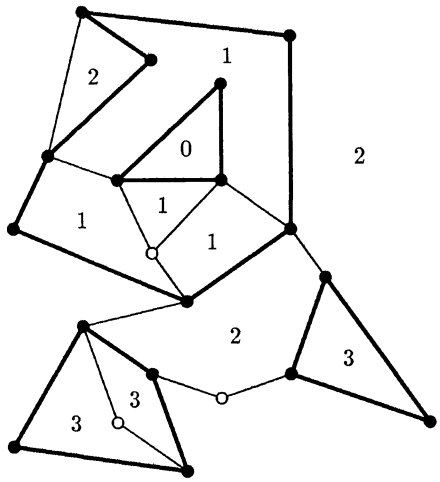


FIG. 3. *An example graph G. Each face is labeled by its type, and the colours of vertices and edges have been indicated, heavy lines signifying black edges.*

Proof. Assume without loss of generality that F_0 lies outside C, and let F be a face inside C of type k. By definition of the types of faces, there are faces

$$F_0, F_1, \cdots, F_{k-1}, F_k = F$$

such that $Type(F_i) = i$, for $i = 0, \cdots, k$, and such that the boundaries of F_i and F_{i+1} have at least one vertex in common, for $i = 0, \cdots, k-1$. Now choose j, $0 \leqq j < k$, such that F_j is outside C, whereas F_{j+1} is inside C (see Fig. 4). Let u be a vertex common to the boundaries of F_j and F_{j+1}. Then $Type(u) = \{j, j+1\} < \{k\}$, and clearly $u \in \mathcal{V}(C)$. □

LEMMA 3. *Let e and e' be distinct black edges with a common endpoint u, and assume that e and e' are not adjacent in the cyclic order around u in $G_{\mathcal{B}}$. Then e and e' do not belong to the same block of $G_{\mathcal{B}}$.*

Proof. Assume the contrary and let C be a simple cycle in $G_{\mathcal{B}}$ containing e and e'. Let $Type(u) = \{k, k+1\}$. As every black edge incident on u has an incident face of type k, both the inside and the outside of C contains a face of type k (see Fig. 5). But since all vertices on C are of type $\{k, k+1\}$, this is impossible by Lemma 2. □

LEMMA 4. *Every block of $G_{\mathcal{B}}$ is a simple cycle.*

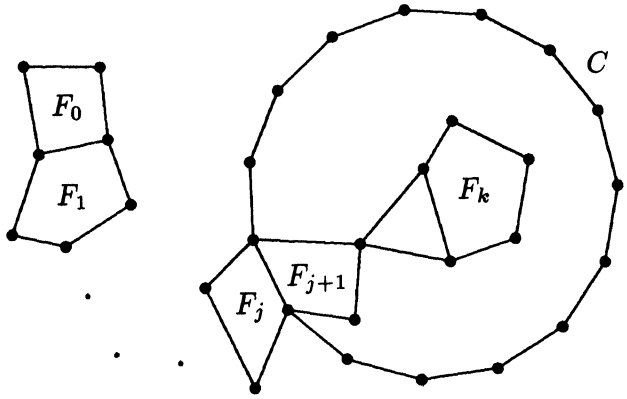


FIG. 4. *A path of faces from the outside to the inside of a simple cycle (Lemma 2).*

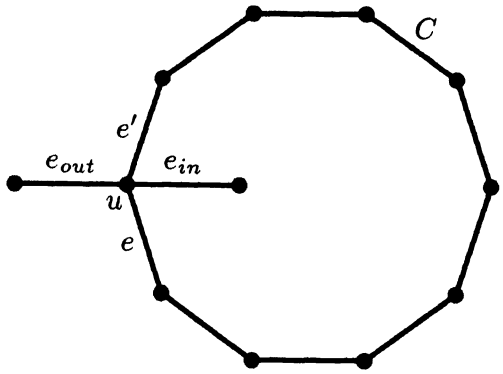


FIG. 5. *A contradiction to Lemma 2:* e_{in} *and* e_{out} *both have an incident face of type k.*

Proof. Let u be a black vertex of type $\{k, k+1\}$ and consider the faces incident on u in G in their cyclic order around u. Their types alternate between k and $k+1$, with a change occurring exactly between those adjacent faces whose common boundary edges are black. Hence all black vertices are of even degree in $G_{\mathcal{B}}$ (the *parity* condition).

Suppose now that B is a block of $G_{\mathcal{B}}$ that is not a simple cycle. Then either B contains just one edge, or there is a vertex of degree $\geqq 3$ in B and therefore, by the parity condition, of degree $\geqq 4$ in $G_{\mathcal{B}}$. But the latter possibility is excluded by Lemma 3. Hence $G_{\mathcal{B}}$ contains a bridge e, and the removal of e splits the connected component of $G_{\mathcal{B}}$ containing e into two connected components. Each of these, by the parity condition, contains exactly one vertex of odd degree. But this is impossible. □

DEFINITION. A *black cycle* is a block of $G_{\mathcal{B}}$. Let $\mathcal{B}$ be the set of black cycles.

It is possible for a single black vertex to belong to several black cycles. Since this situation causes us considerable (mostly notational) difficulty, from now on we will assume that it does not occur, i.e., that each black vertex u belongs to precisely one black cycle, which we will denote by B_u. We later show how to bring about this favourable state of affairs if it does not exist initially.

DEFINITION. Let

$$Q = \{(u, e) \in V \times E \mid e \text{ is incident on } u \text{ in } G \text{ and}$$

$$\exists k \geqq 0\colon \mathit{Type}(u) = \{k, k+1\} \text{ and } \mathit{Type}(e) = \{k+1\}\}.$$

Let $G_{\mathscr{L}}$ be the undirected graph $(V \cup Q, \{\phi(e) \mid e \in E\})$. Here for $e = \{u_1, u_2\} \in E$, $\phi(e) = \{u_1', u_2'\}$, where

$$u_i' = \begin{cases} u_i & \text{if } (u_i, e) \notin Q \\ (u_i, e) & \text{if } (u_i, e) \in Q \end{cases} \quad \text{for } i = 1, 2.$$

Intuitively, Q is a set of "edge attachments" in G, i.e., of pairs $(u, e) \in V \times E$ with e incident on u, and $G_{\mathscr{L}}$ is obtained by breaking each edge attachment belonging to Q and giving all dangling edges new endpoints.

Note that the vertices and some of the edges of G are also vertices and edges, respectively, of $G_{\mathscr{L}}$. These are the *proper* elements of $G_{\mathscr{L}}$. The remaining elements of $G_{\mathscr{L}}$ are called *improper*. Improper vertices have degree one, and each improper edge has at least one improper endpoint. For each improper vertex $q = (u, e) \in Q$, u is called the *origin* of q, and q is called a *representative* of u. We give each representative the same type and colour as its origin, and for all $e \in E$, we give $\phi(e)$ and its directed versions the same type and colour as e. Figure 6 illustrates the various concepts related to $G_{\mathscr{L}}$.

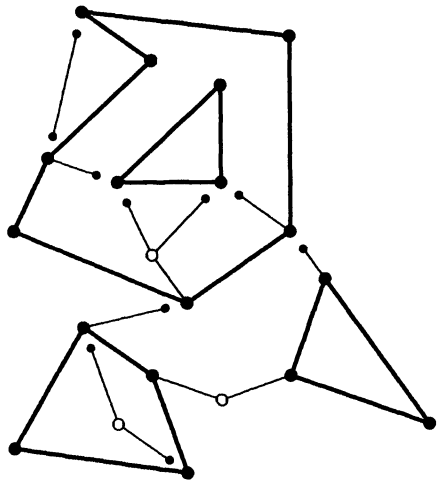


FIG. 6. *The graph $G_{\mathscr{L}}$ corresponding to the example graph G of Fig. 3. Improper vertices are shown smaller than the proper vertices.*

DEFINITION. A *layer* is a connected component of $G_{\mathscr{L}}$. Let $\mathscr{L}$ be the set of layers and for all $u \in V$, let L_u be the layer containing u.

The next lemmas investigate the structure of $G_{\mathscr{L}}$.

LEMMA 5. *For every layer $L \in \mathscr{L}$, there is a unique $k \geqq 0$, called the level of L, such that all vertices and edges in L belong to one of the following categories:*

(1) *Proper black vertices and edges of type $\{k, k+1\}$.*

(2) *Proper white vertices and edges of type $\{k\}$.*

(3) *Improper black vertices of type $\{k-1, k\}$.*

(4) *Improper white edges of type $\{k\}$.*

Proof. The proof is easy. The central observation is that if α_1 and α_2 are proper graph elements of $G_{\mathscr{L}}$ with $Type(\alpha_1) \leqq \{k, k+1\} < Type(\alpha_2)$, for some $k \geqq 0$, then α_1 and α_2 do not belong to the same layer. □

LEMMA 6. *Each block of $G_{\mathscr{L}}$ either is a black cycle or is spanned by a white bridge.*

Proof. It follows from Lemmas 2 and 5 that no simple cycle in $G_{\mathscr{L}}$ can contain a white edge. Therefore all white edges in $G_{\mathscr{L}}$ are bridges. It is clear that if a layer $L \in \mathscr{L}$ contains one edge of a black cycle B, then B is a subgraph of L. Hence each block of $G_{\mathscr{L}}$ that contains a black edge is a black cycle. □

LEMMA 7. *Let u and v be vertices in G that both have a representative in some layer $L \in \mathscr{L}$. Then u and v belong to the same black cycle.*

Proof. Let $Type(u) = Type(v) = \{k, k+1\}$. By the obvious translation of a path in $G_{\mathscr{L}}$, there is a path in G from u to v whose first edge is of type $\{k+1\}$, and none of whose internal vertices lie on B_u. Now clearly $v \in \mathscr{V}(B_u)$, since otherwise B_u would furnish a contradiction to Lemma 2 (see Fig. 7). □

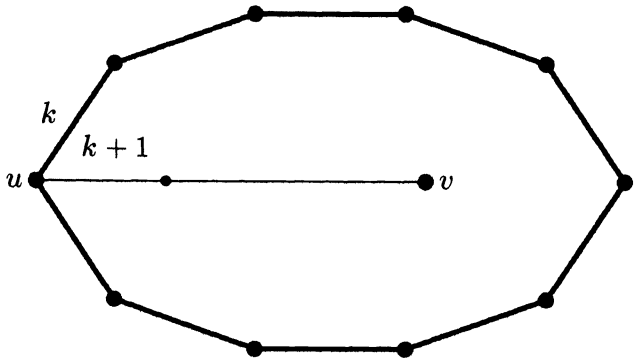


FIG. 7. *A contradiction to Lemma 2: v has an incident face of type k.*

Define the level of a vertex in G or of a black cycle to be the level of the layer containing it, and for $\alpha \in \mathscr{L} \cup V \cup \mathscr{B}$, denote the level of α by $Level(\alpha)$.

LEMMA 8. (a) *$\mathscr{L}$ contains exactly one layer L_0 of level zero. L_0 is the boundary of F_0.*

(b) *Every layer $L \neq L_0$ contains at least one improper vertex.*

Proof. Part (a) is obvious. For part (b), consider a layer $L \in \mathscr{L}$ of level $k \geqq 1$. We may assume that L contains a proper vertex u. Since the type of u is either $\{k\}$ or $\{k, k+1\}$, u has an incident face $F \in \mathscr{F}$ of type k. By definition of the types of faces, some vertex on the boundary of F is of type $\{k-1, k\}$. Hence there is a simple path p on the boundary of F from u to a vertex v of type $\{k-1, k\}$ such that all internal vertices and all edges on p are of type either $\{k\}$ or $\{k, k+1\}$. Now clearly all internal vertices on p belong to L, and v has a representative in L. □

Let $\mathscr{L}^+ = \mathscr{L} \setminus \{L_0\}$. It follows from Lemmas 7 and 8 that for each $L \in \mathscr{L}^+$, there is a unique black cycle, henceforth denoted by $p_{\mathscr{B}}(L)$, that contains all origins of improper vertices in L. For all $L \in \mathscr{L}^+$, let $p_{\mathscr{L}}(L)$ be the layer containing $p_{\mathscr{B}}(L)$. Clearly, $Level(p_{\mathscr{L}}(L)) = Level(L) - 1$. It follows that $T_{\mathscr{L}} = (\mathscr{L}, \{(L, p_{\mathscr{L}}(L)) \mid L \in \mathscr{L}^+\})$ is a directed tree.

Informal explanation. Although we will construct a DFS tree as a static object without explicit reference to an actual depth-first search, it is useful to keep in mind the dynamic process of a search. As described in § 3, the DFS tree constructed by our algorithm will be associated with a depth-first search that always explores black edges before white edges whenever there is a choice. Hence as soon as the search "hits" a black cycle for the first time, it immediately discovers the whole cycle.

A depth-first search satisfying the above condition has a very useful property: It follows from the definition of $p_{\mathscr{B}}$ that when the search is in the maximal subtree T of $T_{\mathscr{L}}$ rooted at some layer $L \in \mathscr{L}^+$ (i.e., it is at a vertex in some layer in T), then it cannot leave T again until all vertices in layers in T have been explored, and when it does so, the search will backtrack to the vertex in $p_{\mathscr{L}}(L)$ from which the search of T was originally begun. Hence the search of each layer L is entirely independent of the search of layers that are descendants of L in $T_{\mathscr{L}}$. Moreover, if $L \neq L_0$, then the search of $p_{\mathscr{L}}(L)$ determines the first vertex in L to be discovered, but does not otherwise influence the search of L. The main outstanding difficulty is to compute for each layer L the first vertex in L to be discovered. Since it turns out to be more convenient, we will instead compute for each black cycle B the first vertex on B discovered by the search, which, as in § 3, we call the *leader* of B.

The following fact is well known and easily checked.

PROPOSITION 9. *Let H be a connected undirected graph, let $B_1, \cdots, B_l$ be the blocks of H, and let s be a vertex in H. For $i = 1, \cdots, l$, let T_i be a* DFS *tree of B_i rooted at the s-dominator of B_i in H. Then $T_1 \cup \cdots \cup T_l$ is a* DFS *tree of H rooted at s.*

By Lemma 6 and Proposition 9, it is easy to compute DFS trees of the layers. Shortly, we will prove a result similar to Proposition 9 (Lemma 10), which implies that suitable DFS trees of the layers can be combined to yield a DFS tree of G.

We now orient the black cycles. Given a black cycle $B \in \mathscr{B}$, let k be the level of B and define $\vec{B}$ as the directed cycle in the directed version of B spanned by the darts of B of type $k+1$ (equivalently, the orientation of $\vec{B}$ is clockwise if and only if F_0 is inside B). For each black vertex u, let $pre(u)$ and $suc(u)$ denote, respectively, the predecessor and the successor of u on $\overrightarrow{B_u}$. For each pair u, v of black vertices with $B_u = B_v$, let $[u, v]$ denote the segment of $\overrightarrow{B_u}$ consisting of the vertices on the simple path in $\overrightarrow{B_u}$ from u to v. Finally, for all $B \in \mathscr{B}$ and all nontrivial segments I of $\vec{B}$, we abbreviate $First_{\vec{B}}(I)$ and $Last_{\vec{B}}(I)$ to $First(I)$ and $Last(I)$, respectively.

DEFINITION. For all $L \in \mathscr{L}^+$ and all vertices $v \in V$ with a representative in L, choose one such representative to be called the *selected representative* of v in L. For each layer $L \in \mathscr{L}^+$ and each vertex v on $p_{\mathscr{B}}(L)$, define the *L-successor* of v to be the selected representative in L of the nearest successor of v on $\overrightarrow{p_{\mathscr{B}}(L)}$ that has a representative in L.

For each black cycle B contained in some layer $L \in \mathscr{L}^+$, let $p_{\mathscr{B}}(B) = p_{\mathscr{B}}(L)$. Let $B_0 = L_0$, $\mathscr{B}^+ = \mathscr{B} \setminus \{B_0\}$, $V_{\mathscr{B}} = \mathscr{V}(\mathscr{B})$ and $V_{\mathscr{B}}^+ = \mathscr{V}(\mathscr{B}^+)$.

DEFINITION. For all black cycles $B \in \mathscr{B}^+$ and all vertices v on $p_{\mathscr{B}}(B)$, let $X_B(v)$ be the q-dominator of B in L, where L is the layer containing B and q is the L-successor of v. Also let $G_{\mathscr{X}}$ be the directed graph $(V_{\mathscr{B}}, E_{\mathscr{X}})$, where

$$E_{\mathscr{X}} = \{(v, u) \in V_{\mathscr{B}} \times V_{\mathscr{B}}^+ \mid v \in \mathscr{V}(p_{\mathscr{B}}(B_u)) \text{ and } X_{B_u}(v) = u\}.$$

Informal explanation. The meaning of $X_B(v)$ is that if v is the leader of $p_{\mathscr{B}}(B)$, then $X_B(v)$ is the leader of B. Consider Fig. 8 to see why this should be true. When the search hits $p_{\mathscr{B}}(B)$ at v, it first explores the rest of $p_{\mathscr{B}}(B)$. We stipulate that this happens in the direction opposite to that given by the orientation of $\overrightarrow{p_{\mathscr{B}}(B)}$. Then the origin of the L-successor q of v represents the first opportunity to discover L, i.e., it is reached after the least amount of backtracking, and we stipulate that q is given preference over any other representative in L of its origin. Finally, if a search of L starts at q, then $X_B(v)$ will become the leader of B.

By the above characterization, the following holds for all black vertices u and v. If there is a path in $G_{\mathscr{X}}$ from v to u and v is the leader of B_v, then u is in turn the leader of B_u. Hence for all $B \in \mathscr{B}$, $X(B)$, as defined below, is the leader of B.

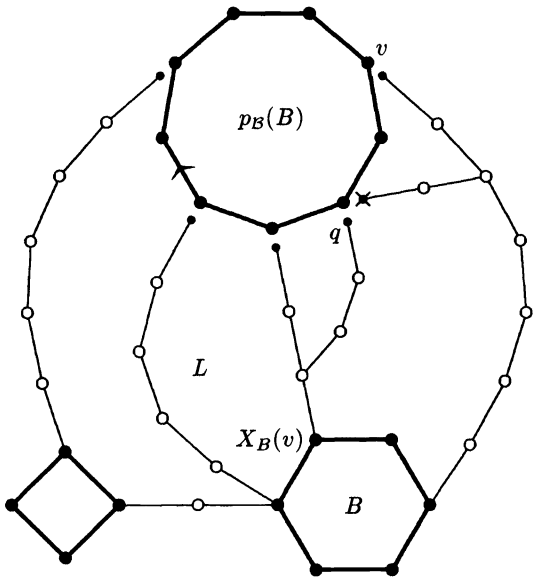


FIG. 8. *If v is the leader of $p_{\mathcal{B}}(B)$, $X_B(v)$ is the leader of B. An arrow indicates the orientation of $\overrightarrow{p_{\mathcal{B}}(B)}$, and improper vertices that are not selected representatives have been marked with a cross.*

DEFINITION. Let $X \subseteq V_{\mathcal{B}}$ be the set of vertices in $G_{\mathcal{B}}$ reachable from r. For all $B \in \mathcal{B}$, let $X(B)$ be the unique vertex in $X \cap \mathcal{V}(B)$ (it follows trivially by induction on the level of B that $|X \cap \mathcal{V}(B)| = 1$).

DEFINITION. Let $L \in \mathcal{L}$. A directed spanning tree T of L is called *consistent* if

(1) If $L = L_0$, then T is rooted at r. If $L \neq L_0$, then T is rooted at the L-successor of $X(p_{\mathcal{B}}(L))$.

(2) For each black cycle B in L, T contains all edges of $\vec{B}$ except $(X(B), suc(X(B)))$.

LEMMA 10. *For all $L \in \mathcal{L}$, let $T_L = (V_L, E_L)$ be a consistent spanning tree of L. Let T be the directed graph on the vertex set V that contains an edge (u, v), for all $u, v \in V$, exactly if for some $L \in \mathcal{L}$, either $(u, v) \in E_L$, or else $(u, q) \in E_L$ for some representative q of v. Then T is a* DFS *tree of G rooted at r (see* Fig. 9).

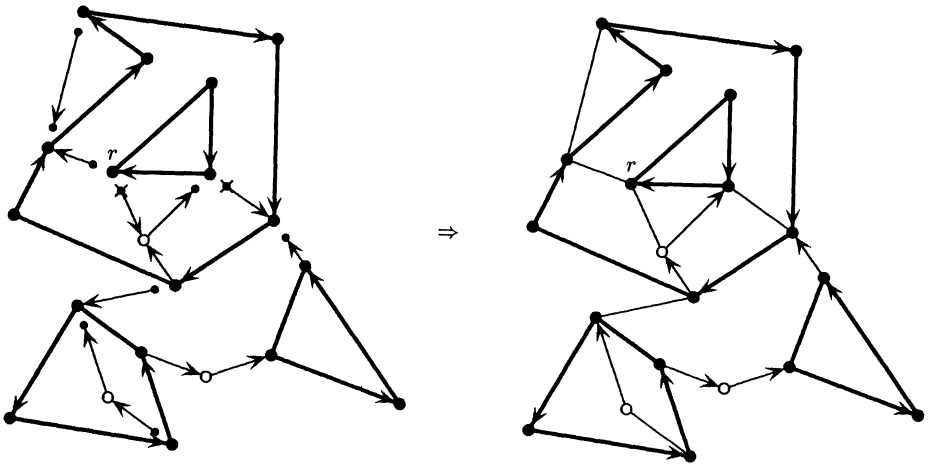


FIG. 9. *The construction of a* DFS *tree of the example graph from consistent spanning trees of its layers.*

Proof. For all vertices $u \in V$, clearly the outdegree of u in T equals its outdegree in T_{L_u}. Hence the outdegree of r in T is zero, and the outdegree in T of each vertex in $V \backslash \{r\}$ is one. A cycle in T must contain vertices belonging to distinct layers. However, every edge in T with endpoints in distinct layers goes from a vertex of level $k+1$ to a vertex of level k, for some $k \geqq 0$. Hence T is acyclic. We may conclude that T is a spanning tree of G rooted at r.

Observe that by Lemma 6 and condition (2) in the definition of consistent spanning trees, T_L is a DFS tree of L, for all $L \in \mathscr{L}$. We finish the proof by showing that G contains no cross edges relative to T. Hence let $e = \{u, v\} \in E$ and consider two cases.

Case 1. $Level(u) = Level(v)$. Then $L_u = L_v$, and since T_{L_u} is a DFS tree of L_u, there is a path p in T_{L_u} from u to v, or vice versa. All edges on p are also edges of T. Hence e is not a cross edge.

Case 2. $Level(u) \neq Level(v)$. Then (after possibly interchanging u and v) $G_{\mathscr{L}}$ contains an edge $\{u, q\}$, where q is a representative of v. Let $L = L_u$, let s be the root of T_L, and put $T' = T_{p_{\mathscr{L}}(L)}$. Since T_L and T' are consistent, the only vertices on $p_{\mathscr{B}}(L)$ that are not ancestors in T' of the origin z of s have no representatives in L (see Fig. 10). Hence v is an ancestor of z in T' and therefore also in T. Finally, u is a descendant of s in T_L, and therefore of z in T. It follows that e is not a cross edge. □

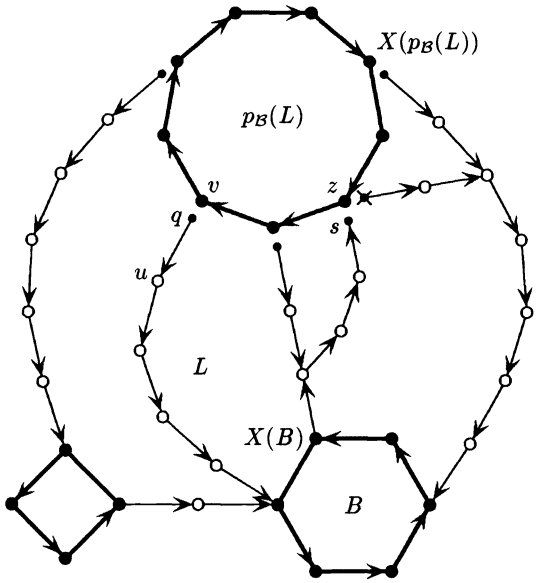


FIG. 10. *Since there is a path in T from u to v, {u, v} is not a cross edge* (*Lemma* 10).

We now turn to the problem of ensuring disjointness of the black cycles. Let us call a black vertex u *shared* if it belongs to more than one black cycle, and *m-fold shared* if the number of black cycles containing u is exactly m. Our approach will be to split each m-fold shared vertex u into $m+1$ pieces, one piece for each black cycle containing u and one central piece connected to the other m pieces, thereby creating a graph $\tilde{G}$ without shared vertices. We then show how a DFS tree of G can be derived very easily from the DFS tree of $\tilde{G}$ computed by our algorithm.

Since the following definition of $\tilde{G}$ is somewhat complicated for technical reasons, whereas the derivation of $\tilde{G}$ from G is conceptually simple, the reader is invited to use the formal definition only to confirm the intuition gained from Figs. 11 and 12.

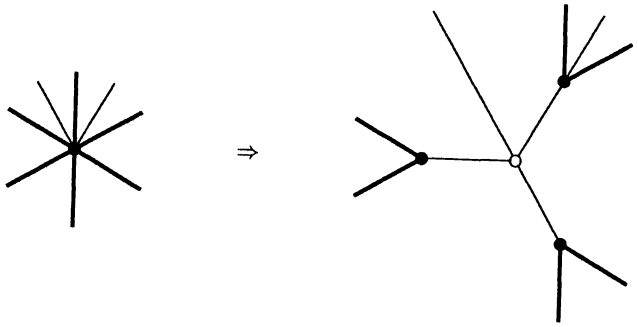


FIG. 11. *A threefold shared vertex is replaced by four vertices.*

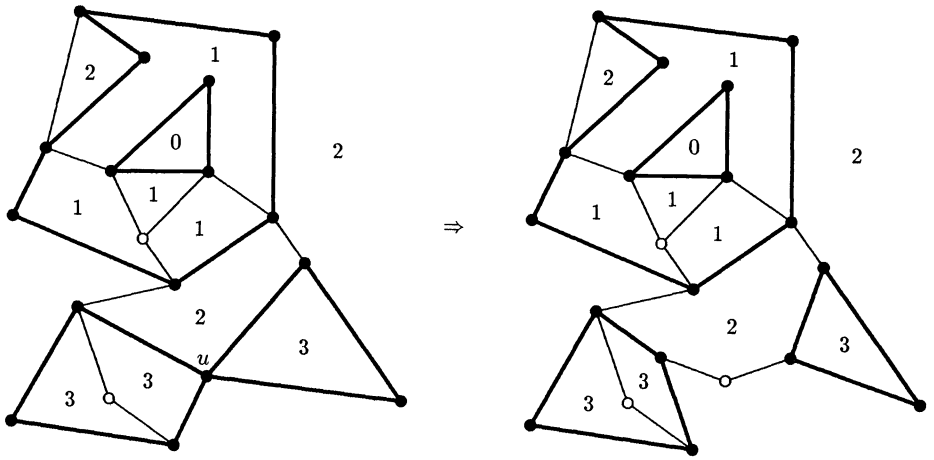


FIG. 12. *The elimination of a shared vertex u in an example graph.*

Let S be the set of shared vertices in G. For all $u \in S$, let $W_u = \{(u, B) \mid B \in \mathcal{B} \text{ and } u \in \mathcal{V}(B)\}$ and put $W = \bigcup_{u \in S} W_u$. For all $B \in \mathcal{B}$ and all darts $(u, v) \in D$, we will say that B *encircles* (u, v) exactly if $(u, B) \in W$, and either $e = \{u, v\}$ is a black edge belonging to B, or else (e is white and) the nearest black predecessor and the nearest black successor of e in the cyclic order around u both belong to B.

Now let $\tilde{G}$ be the undirected graph $(V \cup W, \{\tilde{e} \mid e \in E\} \cup \tilde{E})$. Here $\tilde{E} = \{\{u, (u, B)\} \mid (u, B) \in W\}$, and for $e = \{u_1, u_2\} \in E$, $\tilde{e} = \{u_1', u_2'\}$, where for $i = 1, 2$, $u_i' = u_i$ if (u_i, u_{3-i}) is not encircled by any black cycle, and $u_i' = (u_i, B)$ otherwise, where B is the unique black cycle encircling (u_i, u_{3-i}).

For all $\{u, v\} \in E$, there clearly is a path in $\tilde{G}$ from u to v all of whose internal vertices belong to $W_u \cup W_v$. It follows easily that $\tilde{G}$ is biconnected. We will consider $\tilde{G}$ to be embedded in the plane in a natural way illustrated in Figs. 11 and 12. While we will not spell out the tedious details of this planar embedding, we note that its existence is guaranteed by Lemma 3. Let $\tilde{G}_D$ be the face incidence graph of $\tilde{G}$. Note that there is a one-to-one correspondence between the faces of G and those of $\tilde{G}$ such that if the face of $\tilde{G}$ corresponding to a face F of G is denoted by $\tilde{F}$, then:

(1) For all $F \in \mathcal{F}$, if E_F is the set of edges on the boundary of F, then the edges on the boundary of $\tilde{F}$ are those in the set $\{\tilde{e} \mid e \in E_F\}$, possibly together with some edges in $\tilde{E}$.

(2) Each edge in $\tilde{E}$ is incident on two faces $\widetilde{F_1}$ and $\widetilde{F_2}$ of $\tilde{G}$ with the property that F_1 and F_2 have the same type in G.

For all $F \in \mathcal{F}$, let $\widetilde{Type}(\tilde{F})$ be the distance in $\tilde{G}_D$ from $\widetilde{F_0}$ to $\tilde{F}$. We will show that $\widetilde{Type}(\tilde{F}) = Type(F)$ for all $F \in \mathcal{F}$. Note first that for all $F_1, F_2 \in \mathcal{F}$, if $\widetilde{F_1}$ and $\widetilde{F_2}$ are adjacent in $\tilde{G}_D$, then F_1 and F_2 are adjacent in G_D. Hence $\widetilde{Type}(\tilde{F}) \geqq Type(F)$ for all $F \in \mathcal{F}$. On the other hand, if $Type(F_1) < Type(F_2)$ and F_1 and F_2 are adjacent in G_D, but $\widetilde{F_1}$ and $\widetilde{F_2}$ are not adjacent in $\tilde{G}_D$, then the boundaries of F_1 and F_2 share a vertex in S, in which case $\widetilde{F_2}$ is adjacent in $\tilde{G}_D$ to a face $\tilde{F}$ with $Type(F) = Type(F_1)$. Hence by induction, $\widetilde{Type}(\tilde{F}) \leqq Type(F)$ for all $F \in \mathcal{F}$.

We now need to apply the concepts of initial face, edge colours, layers, and shared vertices to $\tilde{G}$ as well as to G. The clause "(in $\tilde{G}$)" will be used in otherwise ambiguous cases to indicate when these concepts should be understood as defined relative to $\tilde{G}$. Choose F_0 as the initial face of $\tilde{G}$. By what we have just shown, each edge in $\tilde{E}$ is white, and for all $e \in E$, the colour of $\tilde{e}$ (in $\tilde{G}$) is the same as the colour of e (in G). It follows easily that no vertex in $\tilde{G}$ has more than two incident black edges (in $\tilde{G}$), i.e., $\tilde{G}$ has no shared vertices.

All that remains is to show that a DFS tree of G can be obtained from a DFS tree of $\tilde{G}$ computed by our algorithm. First define $Source(u) = u$ for all $u \in V$ and $Source(w) = u$ for all $u \in S$ and all $w \in W_u$.

LEMMA 11. *Let $T = (V \cup W, E_T)$ be a* DFS *tree of $\tilde{G}$ rooted at r and of the form described in Lemma* 10, *and let T' be the directed graph*

$$(V, \{(Source(u), Source(v)) \in V \times V \mid (u, v) \in E_T \text{ and } Source(u) \neq Source(v)\}).$$

Then T' is a DFS *tree of G rooted at r.*

Proof. Let $e = \{u, v\} \in \tilde{E}$ and observe first that u and v belong to the same layer (in $\tilde{G}$), since otherwise we would have a contradiction to Lemma 7 (see Fig. 11). We next show that a directed version of e is contained in T. Since e is not a cross edge relative to T, T must contain a path p from u to v, or vice versa. Assume that p contains more than one edge. Then, since e is a bridge in its layer L (in $\tilde{G}$), p must contain a vertex outside L. But T is assumed to be of the form described in Lemma 10; hence there can be no such path.

We now know that for all $u \in S$, there is precisely one vertex among $\{u\} \cup W_u$ whose parent in T does not exist or does not belong to $\{u\} \cup W_u$. It follows that the outdegree of each vertex in T' is one, except that the outdegree of r is zero. Furthermore it is easy to see that for all $u, v \in V \cup W$, if there is a path in T from u to v, then there is a corresponding path in T' from $Source(u)$ to $Source(v)$. Hence T' is a DFS tree of G rooted at r. □

5. A first DFS tree construction algorithm. We assume a standard adjacency list representation of directed and undirected graphs, which we will not specify completely. The representation of a graph consists of a list of its vertices together with an *adjacency list* for each vertex. For each vertex u in the graph, each edge incident on u is represented by one element in u's adjacency list. All objects that are explicitly manipulated are represented by unique integer *names.*

To be able to state the central part of the algorithm reasonably succinctly, we first tackle a number of subroutines to be used later, sometimes without explicit reference. Recall that our model of computation is the PRIORITY PRAM.

LEMMA 12 [3] (Prefix sums). *Given n integers $a_1, \cdots, a_n$ of size $n^{O(1)}$, the quantities $\sum_{i=1}^{k} a_i$, for $k = 1, \cdots, n$, can be computed in $O(\log n / \log \log n)$ time with $O(n \log \log n / \log n)$ processors.*

A frequent use of parallel prefix sums computations is to compact a data structure spread over a large array into a smaller array to enable more efficient access to the data structure. Assuming the data structure to be divided into records of size $O(1)$, associate with each such record a number equal to its size, associate with each array element that is not part of the data structure a zero, and carry out a prefix sums computation on the numbers thus defined. As a result, the records of the data structure receive distinct integer values that may be interpreted as positions in a compacted array. In case the data structure contains pointers into itself, these can easily be adjusted before the records are moved to their new positions.

LEMMA 13 (Pointer doubling). *The following problem can be solved in $O(\log n)$ time with $O(n/\log n)$ processors. Given a directed forest $F=(V, E)$ on n vertices, a set R, a label $\lambda(u) \in R$ for each $u \in V$, and an associative operation $\circ: R \times R \to R$ that can be evaluated on specific arguments in constant time by one processor, compute for all $u \in V$ the value*

$$\Lambda(u) = \lambda(u) \circ \lambda(u_1) \circ \cdots \circ \lambda(u_k),$$

where $(u, u_1, \cdots, u_k)$ is the path in F from u to the root of the tree in F containing u.

Proof. First trivially handle and remove all trees in F with at most six vertices. Then compute an independent vertex set S' in F that contains at least $\frac{1}{6}$ of the vertices in F. This can be done in $O(\log n/\log\log n)$ time with $O(n \log\log n/\log n)$ processors (see [12]). Let S be the set of nonroot vertices in S', i.e., the set of vertices in S' of outdegree one in F. Next, for each child $u \in V$ of a vertex $v \in S$, u is made a child of the parent of v, and u's label is replaced by $\lambda(u) \circ \lambda(v)$. Finally, all vertices in S and their incident edges are removed, and a prefix sums computation is used to "close up the gaps" left in the representation of F by vertices and edges that were removed. This reduces the number of vertices by at least a constant factor. Let $F'=(V \backslash S, E')$ be the resulting forest. It is clear that for all $u \in V \backslash S$, $\Lambda(u)$ computed with respect to F' is the same as $\Lambda(u)$ computed with respect to F. Furthermore, for each $u \in S$, $\Lambda(u)$ can be computed as $\lambda(u) \circ \Lambda(v)$, where v is the parent of u in F (which is not in S). Since the problem becomes easy once the number of vertices has been reduced to $O(n/\log n)$, the desired conclusion now follows by an application of Lemma 1 of [12]. □

Pointer doubling has numerous applications. If each root is labeled by zero, all other vertices are labeled by one, and $\circ$ denotes addition, we compute the depth of each vertex in its tree. The special case of this computation in which F is a path is known as *list ranking*. If each vertex is labeled by its own name and $u \circ v = v$ for all $u, v \in V = R$, the root of the tree containing u is found for all $u \in V$. Given a directed simple cycle C containing a number of "distinguished" elements, we may remove each edge leaving a distinguished element and use the above setting to compute for each vertex on C its nearest distinguished successor. This in turn may be used to construct the adjacency lists of graphs such as the subgraph of a given graph spanned by a set of marked edges.

LEMMA 14. *The following problems can all be solved in $O(\log n)$ time with $O(n/\log n)$ processors*:

(a) (Connected components) *Given an undirected planar graph $G=(V, E)$ on n vertices, compute its connected components, i.e., name the connected components of G and mark each vertex $u \in V$ with the name of the connected component containing u.*

(b) (Spanning forest) *Given an undirected planar graph $G=(V, E)$ and a set $\{r_1, \cdots, r_l\} \subseteq V$ containing precisely one vertex from each connected component of G, compute a graph $T_1 \cup \cdots \cup T_l$, where T_i, for $i=1, \cdots, l$, is a directed spanning tree, rooted at r_i, of the connected component of G containing r_i.*

(c) (Biconnected components) *Given an undirected planar graph $G=(V, E)$ on n vertices, compute its blocks, i.e., name the blocks of G and mark each edge $e \in E$ with the name of the block containing e.*

(d) *Given an embedded undirected planar graph G on n vertices, construct an embedded graph whose connected components are isomorphic to the blocks of G, together with the relevant isomorphisms.*

(e) (Expression evaluation) *Given an n-vertex undirected planar graph $G=(V, E)$ and a label $\lambda(u) \in \mathbb{N}$ for each $u \in V$, compute for each connected component of G the maximum label of a vertex in the component.*

Proof. Part (a) was proved in [12].

(b) Choose r with $r \notin V$ and construct the connected graph $G'=(V \cup \{r\}, E \cup \{\{r, r_i\} \mid 1 \leqq i \leqq l\})$. Compute an undirected spanning tree T of G' [12] and use optimal list ranking to convert T to a directed spanning tree T' of G' rooted at r, as described in [19]. Finally, obtain $T_1 \cup \cdots \cup T_l$ by removing from T' the vertex r and its incident edges.

(c) Use part (a) to compute a set $\{r_1, \cdots, r_l\}$ containing precisely one vertex from each connected component of G. Then construct G' as in part (b) and compute the blocks of G' [12]. The blocks of G are those of G', except for those spanned by the bridges $\{r, r_i\}$, $i=1, \cdots, l$.

(d) We assume that the planar embedding of G is given via the ordering of its adjacency lists: For each vertex u, the order of the edges incident on u in its adjacency list is the same as their cyclic order around u in the embedding (take the first list element to be the successor of the last element).

First mark each edge in G with the name of its block as in part (c). Then, for each articulation point u and each block B containing u, create a new vertex (u, B). The remaining problem is to construct the adjacency lists of the new vertices from the original adjacency lists of G's articulation points. Note that a vertex of the form (u, B) should receive those elements of u's adjacency list that represent edges in B, and that the new graph will inherit an embedding from the embedding of G if the relative order of these elements is preserved. Observe, finally, that if e_1, e_2, e_3, and e_4 are edges in G incident on a common vertex u, with e_1 and e_3 belonging to the same block B and e_2 and e_4 belonging to a different block B', then the cyclic order of the four edges around u in G cannot be (e_1, e_2, e_3, e_4). This follows from the fact that there are simple cycles in G through (i.e., containing) e_1 and e_3, and through e_2 and e_4, but not through e_1 and e_2. Hence if we first use list ranking and a prefix sums computation to move the elements of each adjacency list to consecutive positions in an array, without altering their order, we are left with an instance of the following more abstract problem:

> Given a sequence of n symbols $Z_1, \cdots, Z_n$ such that for all i, j, k, l with $1 \leqq i < j < k < l \leqq n$, it is not the case that $Z_i = Z_k \neq Z_j = Z_l$ (the *nesting* property), determine for $i=1, \cdots, n$ the smallest j, $i < j \leqq n$, such that $Z_j = Z_i$, or determine that no such j exists.

CLAIM. The above problem can be solved in $O(\log n)$ time with $O(n/\log n)$ processors.

Proof. Let $U=\{1, \cdots, n\}$ and let $Y=\{Z_i \mid i \in U\}$ be the set of used symbols. For all $y \in Y$, put $First(y)=\min\{i \in U \mid Z_i = y\}$ and $Last(y)=\max\{i \in U \mid Z_i = y\}$ and let $S=\{First(y) \mid y \in Y\}$ and $T=\{Last(y) \mid y \in Y\}$. Our task is, for all $i \in U \setminus T$, to compute $N(i)=\min\{j > i \mid Z_j = Z_i\}$.

Construct the directed graph $H=(U, E_H)$, where E_H consists of all edges $(i, j) \in U \times U$ for which $i \neq j$ and either

(1) $j=i+1$, $i \in T$ and $j \in S$, or

(2) For some $y \in Y$, $i = First(y)$ and $j = Last(y)$.

It is easy to see that H is a sum of disjoint paths (see Fig. 13). For all $i \in U$, let $F(i)$ be the last vertex on the path containing i, and for all $i \in U \setminus T$, let

$$N'(i) = \begin{cases} i+1 & \text{if } Z_{i+1} = Z_i, \\ F(i+1)+1 & \text{if } Z_{i+1} \neq Z_i. \end{cases}$$

We will show that $N'(i) = N(i)$ for all $i \in U \setminus T$. Since this is obvious if $Z_{i+1} = Z_i$, let us assume that $i \in U \setminus T$ and that $Z_{i+1} \neq Z_i$.

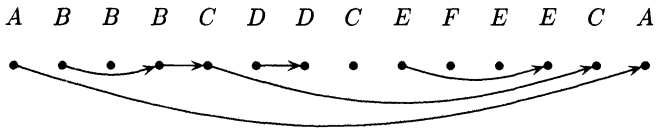


FIG. 13. *An example sequence with the nesting property and the corresponding graph H.*

On the one hand, $N'(i) \leqq N(i)$ since no edge of H can leave the vertex set $W = \{i+1, \cdots, N(i)-1\}$, i.e., be of the form (j, k), with $j \in W$ and $k \in U \setminus W$. On the other hand, $N'(i) \geqq N(i)$. To see this, let p be the path in H containing $i+1$ and observe that if $j \in \mathcal{V}(p) \cap T$ and $i+1 \leqq j < N(i)-1$, then $j+1 \in S$ and hence $(j, j+1) \in E_H$. This implies that $F(i+1) \geqq N(i)-1$, from which the desired conclusion follows.

We have shown how to compute $N(i)$ for all $i \in U \setminus T$. $First(y)$ and $Last(y)$ are easily determined for all $y \in Y$ by the PRIORITY PRAM, and the only other nontrivial step, the computation of $F(i)$ for all $i \in U$, can be carried out by means of pointer doubling. This ends the proof of the claim and of part (d) of the lemma.

(e) First construct G' as in part (c) and compute a directed spanning tree of G' rooted at r. We may now use a slight modification of the optimal expression evaluation algorithm given by Gibbons and Rytter [7]. For more details, see Lemma 9 of [12]. □

LEMMA 15. *Given a directed or undirected graph $G = (V, E)$ on n vertices, the distance in G from u to v can be computed for all $u, v \in V$ in $O(\log n)$ time with n^3 processors.*

Proof. If G is undirected, replace G by its directed version. Assume for simplicity that $V = \{0, \cdots, n-1\}$. After the execution of the algorithm below, the distance $d(i, j)$ from i to j can be found in $A[i, j]$, for all $i, j \in V$. B is an auxiliary array. The meaning of the construct

for i, j, k **in** $\{0, \cdots, n-1\}$ **pardo** S

is that the statement S should be executed for each value of (i, j, k) in $\{0, \cdots, n-1\}^3$ by exactly one processor P_{ijk}. For all $i, j \in V$, we require that if $0 \leqq k_1 < k_2 \leqq n-1$, then P_{ijk_1} should have higher priority (i.e., a lower processor number) than P_{ijk_2}. The meaning of

for i, j **in** $\{0, \cdots, n-1\}$ **pardo** S

is similar, except that all but n^2 processors remain idle. We assume that the processors are synchronized at each acesss to A and B, but omit the necessary synchronization details.

(01) **for** i, j **in** $\{0, \cdots, n-1\}$ **pardo** $A[i, j] := \begin{cases} 0 & \text{if } i = j, \\ 1 & \text{if } (i, j) \in E \text{ and } i \neq j, \\ \infty & \text{if } (i, j) \notin E \text{ and } i \neq j; \end{cases}$

(02) **for** $t := 1$ **to** $\lceil \log n \rceil$ **do**
(03) **for** i, j, k **in** $\{0, \cdots, n-1\}$ **pardo**
(04) **begin**
(05) $B[i, j, k] := 0$;
(06) **if** $A[i, k] + A[k, j] < n$
(07) **then** $B[i, j, A[i, k] + A[k, j]] := 1$;
(08) **if** $B[i, j, k] = 1$
(09) **then** $A[i, j] := k$;
(10) **end**;

The algorithm works by repeated matrix squaring. Lines (05)-(09), which together implement the instruction

$$A[i, j] := \min_{0 \leqq k \leqq n-1} (A[i, k] + A[k, j]),$$

implicitly use the capabilities of the PRIORITY PRAM to compute the minimum of n numbers in unit time. It is easy to see by induction that after the tth execution of the loop in lines (03)-(10), $A[i, j] = d(i, j)$ for all $i, j \in V$ with $d(i, j) \leqq 2^t$, for $t = 0, \cdots, \lceil \log n \rceil$. Hence the algorithm correctly computes $d(i, j)$ for all $i, j \in V$ in $O(\log n)$ time. □

Remark. Lemma 15 remains true if the PRIORITY PRAM is replaced by the weaker COMMON PRAM (see [5, Thm. 4]). This, however, is of little consequence here since we seem to need the PRIORITY PRAM in the algorithm of Lemma 14(d).

We now come to our main results.

THEOREM 1. *Given an undirected connected planar graph $G = (V, E)$ on n vertices and a vertex $r \in V$, a* DFS *tree of G rooted at r can be computed in $O(\log n)$ time by a* PRIORITY PRAM *with n^3 processors.*

Proof. Use the algorithm consisting of steps (1)-(17) below. We adopt the same notation as in the previous section. Steps for which nothing else is stated can be carried out in $O(\log n)$ time with $O(n/\log n)$ processors.

(1) Embed G in the plane. By the result of [14], this can be done in $O(\log n)$ time with n processors, and possibly with fewer processors. Another algorithm is given in [10].
(2) For each block B of G, compute the r-dominator of B in G. This is easy using parts (b) and (c) of Lemma 14: Given any directed spanning tree T of G rooted at r and a block B of G, the r-dominator of B in G is the unique vertex in G that has an incident edge belonging to B, but whose father pointer (if it exists) is not a directed version of an edge in B.
(3) Construct an embedded graph G' whose connected components are isomorphic to the blocks of G, together with the relevant isomorphisms (Lemma 14(d)).

For the sake of simplicity, we will assume in the remaining description that G is biconnected and that $G' = G$. The reader should keep in mind, however, that in reality the computation described in steps (4)-(17) takes place simultaneously for all connected components of G'. We omit the description of a final step in which the DFS trees of the components of G' are combined as described in Proposition 9 to yield a DFS tree of the input graph. We also assume that $|V| \geqq 3$.

(4) Construct the face-vertex incidence graph $G'_{\mathscr{D}}$ of G. This is the undirected bipartite graph on the vertex sets V and $\mathscr{F}$ that contains an edge $\{u, F\}$, for all $u \in V$ and $F \in \mathscr{F}$, exactly if F is incident on u in G. $G'_{\mathscr{D}}$ is computed as follows. Use the planar embedding of G to construct the face cycle graph H of G. Execute a connected components algorithm on the undirected version of H in order to obtain a name for each face and to mark each dart in D with the name of its incident face. Finally, for each dart $(u, v) \in D$, include in $G'_{\mathscr{D}}$ the edge $\{u, F\}$, where F is the face incident on (u, v). Since for each vertex α in $G'_{\mathscr{D}}$, precisely one edge incident on α is generated either for each dart on some cycle in H (if $\alpha \in \mathscr{F}$) or for each element of α's adjacency list in G (if $\alpha \in V$), it is easy to construct the adjacency lists of $G'_{\mathscr{D}}$. Note that $G'_{\mathscr{D}}$ is planar and that by Euler's formula, it contains at most $3n$ vertices.

(5) Choose a face F_0 incident on r and compute the distance in $G'_{\mathscr{D}}$ from F_0 to every other face. By Lemma 15, this can be done in $O(\log n)$ time with n^3 processors.

(6) Compute $Type(\alpha)$ for all $\alpha \in V \cup E \cup D \cup \mathscr{F}$. The type of a face is its distance from F_0 in the face incidence graph $G_{\mathscr{D}}$, which may clearly be computed as half its distance from F_0 in $G'_{\mathscr{D}}$.

(7) Construct $G_{\mathscr{B}}$.

(8) Compute the blocks of $G_{\mathscr{B}}$, i.e., the black cycles.

(9) Construct $\tilde{G}$, substitute it for G and redo previous steps as necessary.

(10) Construct $G_{\mathscr{L}}$.

(11) Determine the connected components of $G_{\mathscr{L}}$, i.e., the layers.

(12) For each $L \in \mathscr{L}^+$, choose the selected representatives in L and determine $p_{\mathscr{B}}(L)$.

(13) For all $u \in V_{\mathscr{B}}$, find $suc(u)$.

(14) Compute the set X and determine the L-successor of $X(p_{\mathscr{B}}(L))$, for all $L \in \mathscr{L}^+$. First compute the L-successor of v for all layers $L \in \mathscr{L}^+$ and all vertices $v \in p_{\mathscr{B}}(L)$. This can be done via pointer doubling in $O(\log n)$ time with n processors per layer, a total of $O(n^2)$ processors (one might think that $O(n)$ processors would suffice for this substep. However, since $p_{\mathscr{B}}$ is not necessarily injective, this is not the case, and the size of the output may be $\Omega(n^2)$). Then determine the distance between each pair of vertices in $G_{\mathscr{L}}$. By Lemma 15, $O(\log n)$ time and n^3 processors suffice. Using the information computed so far, find $X_B(v)$, for each $B \in \mathscr{B}^+$ and each vertex $v \in p_{\mathscr{B}}(B)$, as the vertex on B closest in $G_{\mathscr{L}}$ to the L-successor of v, where L is the layer containing B. It is now easy to construct the graph $G_{\mathscr{X}}$. By another application of Lemma 15, the set X of vertices in $G_{\mathscr{X}}$ reachable from r can be computed in $O(\log n)$ time with n^3 processors.

(15) For each layer $L \in \mathscr{L}$, construct a consistent spanning tree of L. This is trivial for $L = L_0$. For $L \neq L_0$, let r_L be the L-successor of $X(p_{\mathscr{B}}(L))$, which was computed in step (14). Construct a directed spanning tree T_L of L rooted at r_L and modify it as follows: For each black cycle B contained in L, remove from T_L all edges in the directed version of B and replace them with all edges of $\vec{B}$ except $(X(B), suc(X(B)))$. It is easy to see that the resulting graph is still a directed tree, and hence a consistent spanning tree of L.

(16) Combine the consistent spanning trees computed in step (15) to a DFS tree of $\tilde{G}$ as described in Lemma 10.

(17) Convert the DFS tree of $\tilde{G}$ computed in step (16) into a DFS tree of G as described in Lemma 11.

Correctness of the algorithm follows from the results of § 4. We have argued that steps (1), (5), and (14) can be executed in $O(\log n)$ time with n^3 processors, and all other steps can be executed in $O(\log n)$ time with $O(n/\log n)$ processors. □

6. Reducing the number of processors. In this section we describe a more complicated but also more processor-economical computation of X. One reason the naive approach of the previous section fails to achieve optimality is that $G_{\mathscr{X}}$ may have $\Omega(n^2)$ edges. It does, however, have a nice property that we will be able to exploit: For all $u \in V_{\mathscr{B}}^+$, the vertices $v \in V_{\mathscr{B}}$ with $(v, u) \in E_{\mathscr{X}}$ form a segment of $\overrightarrow{p_{\mathscr{B}}(B_u)}$. Before we prove this statement, let us introduce some convenient notation.

DEFINITION. For all $u \in V_{\mathscr{B}}^+$, let $f(u) = \{v \in V_{\mathscr{B}} \mid (v, u) \in E_{\mathscr{X}}\}$, and let $pre_{f \neq \emptyset}(u)$ denote the nearest predecessor v of u on $\overrightarrow{B_u}$ with $f(v) \neq \emptyset$. For all $u \in \mathscr{V}(B_0)$, let $f(u) = \{u\}$.

DEFINITION. For all $u \in V_{\mathscr{B}}^+$, let G_u be the connected component containing u of the graph obtained from $G_{\mathscr{L}}$ by the removal of both black edges incident on u, and let

$$Y(u) = \{v \in \mathscr{V}(p_{\mathscr{B}}(B_u)) \mid v \text{ is the origin of a selected representative in } \mathscr{V}(G_u)\}.$$

For all $B \in \mathscr{B}^+$, let $Y(B) = \bigcup_{u \in \mathscr{V}(B)} Y(u)$.

LEMMA 16. *Let $u \in V_{\mathscr{B}}^+$. Then $Y(u)$ is a segment of the cyclic order of $Y(B_u)$ on $\overrightarrow{p_{\mathscr{B}}(B_u)}$.*

Proof. For all $v \in \mathscr{V}(B_u)$ with $v \neq u$, G_u and G_v are vertex-disjoint since otherwise there would be a cycle in $G_{\mathscr{L}}$ containing a white edge. The result now follows by planarity considerations (see Fig. 14). □

DEFINITION. For all $u \in V_{\mathscr{B}}^+$ such that $Y(u)$ is a nontrivial segment of the cyclic order C_u of $Y(B_u)$ on $\overrightarrow{p_{\mathscr{B}}(B_u)}$, let $Last(u) = Last_{C_u}(Y(u))$.

LEMMA 17. *Let $u \in V_{\mathscr{B}}^+$. Then*

(a) $Y(u) = \emptyset \Leftrightarrow f(u) = \emptyset$.

(b) $Y(u) = Y(B_u) \Leftrightarrow f(u) = \mathscr{V}(p_{\mathscr{B}}(B_u)) \Leftrightarrow pre_{f \neq \emptyset}(u) = u$.

(c) *If $\emptyset \subset Y(u) \subset Y(B_u)$, then $f(u) = [Last(pre_{f \neq \emptyset}(u)), pre(Last(u))]$.*

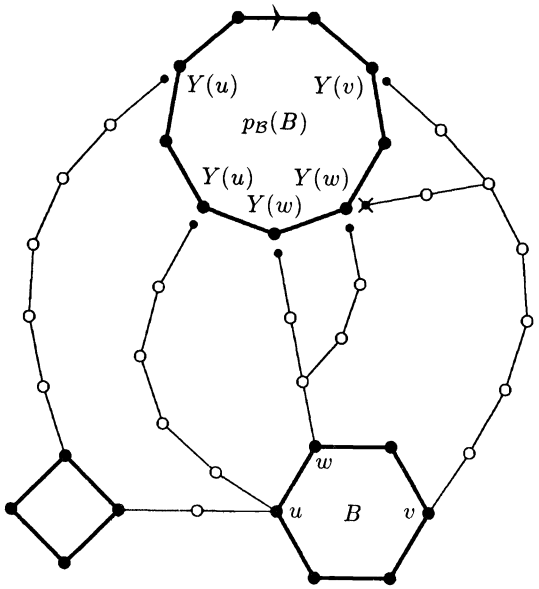


FIG. 14. *$Y(u)$ is a segment of the cyclic order of $Y(B_u)$. For $\alpha \in \{u, v, w\}$, the elements of $Y(\alpha)$ are marked "$Y(\alpha)$."*

Proof. Observe that if q is a selected representative in L_u, then u is the q-dominator of B_u exactly if $q \in \mathcal{V}(G_u)$. Hence

$$f(u) = \{v \in \mathcal{V}(p_{\mathcal{B}}(B_u)) \mid \text{the nearest successor of } v \text{ in } Y(B_u) \text{ belongs to } Y(u)\}.$$

Now (a) and (b) are obvious. Part (c) follows by planarity from the same characterization (see Fig. 15). □

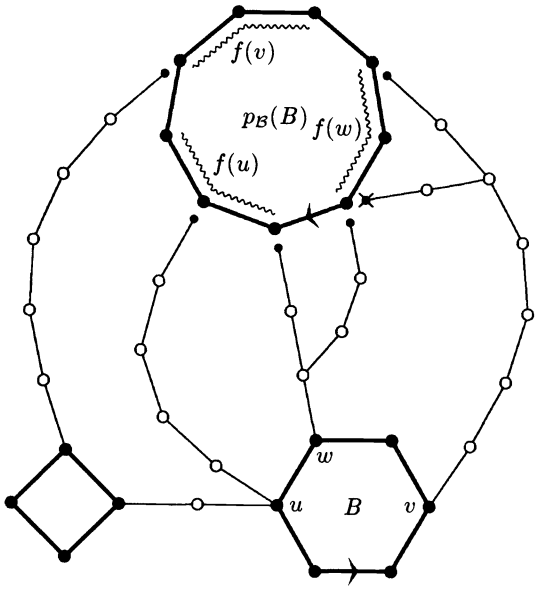


FIG. 15. *A monotonicity property of* f (*Lemma* 17(c)). *For* $\alpha \in \{u, v, w\}$, $f(\alpha)$ *is indicated by a wavy line "spanning"* $f(\alpha)$.

LEMMA 18. $f(u)$ *can be computed for all* $u \in V_{\mathcal{B}}$ *in* $O(\log n)$ *time with* $O(n/\log n)$ *processors.*

Proof. With $G_{\mathcal{L}}$ embedded in the obvious way, compute its face cycles by means of pointer doubling. When we refer to face cycles and darts below, we mean those of $G_{\mathcal{L}}$.

We begin by showing that all white darts in a given layer L lie on the same face cycle, which we will call the *outer face cycle* of L. First observe that if a face cycle contains some white dart (u, v), then it must also contain (v, u), since otherwise suitable "shortcutting" operations would yield a simple cycle in $G_{\mathcal{L}}$ containing $\{u, v\}$. It follows that if two white darts have a common endpoint, then they lie on the same face cycle. It is now easy to see that the same holds for two white darts that have an endpoint each on a common black cycle, from which the above claim follows.

For each outer face cycle, choose an arbitrary dart on the cycle and label by means of list ranking each dart e on the cycle with the distance from e to the chosen dart. This allows the cyclic order of three given darts on a common outer face cycle to be determined in constant time. Also compute by means of pointer doubling for each dart e on an outer face cycle the nearest successor dart $\sigma(e)$ on the cycle whose tail is a selected representative.

Now let $u \in V_{\mathcal{B}}^+$. If u has no incident white edges in $G_{\mathcal{L}}$, clearly $Y(u) = \emptyset$. Otherwise, let e_1 and e_2 be, respectively, the first white dart and the first black dart in the cyclic order around u in $G_{\mathcal{L}}$. Then the following holds (see Fig. 16):

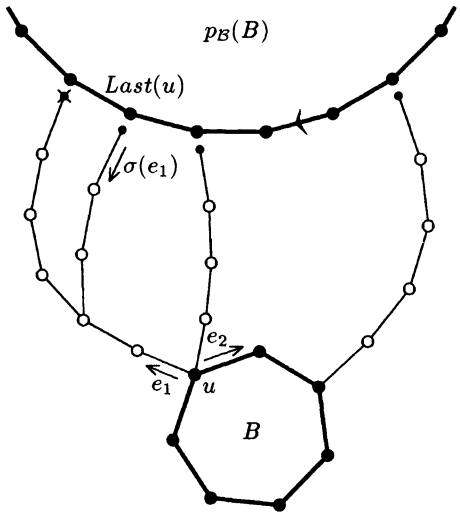


FIG. 16. *The computation of* $Last(u)$.

(1) If $\sigma(e_1)$ occurs after e_2, i.e., if the cyclic order of e_1, e_2, and $\sigma(e_1)$ on the outer face boundary of L_u is $(e_1, e_2, \sigma(e_1))$, then $\mathcal{V}(G_u)$ does not contain any selected representatives, i.e., $Y(u) = \emptyset$.

(2) If $\sigma(e_1)$ occurs before e_2, i.e., if the cyclic order of e_1, e_2, and $\sigma(e_1)$ on the outer face boundary of L_u is $(e_1, \sigma(e_1), e_2)$, then the tail of $\sigma(e_1)$ is a selected representative in G_u.

In case (2), we can say more. Let $q_1, \cdots, q_l$ be the selected representatives in L_u, and for $i = 1, \cdots, l$, let v_i be the origin of q_i, and let e_i be the dart in $G_{\mathcal{L}}$ with tail q_i. If the cyclic order of $e_1, \cdots, e_l$ on the outer face cycle of L_u is $(e_1, \cdots, e_l)$, then the cyclic order of the origins of their tails on $\overrightarrow{p_{\mathcal{B}}(B_u)}$ is $(v_l, \cdots, v_1)$. We may conclude that in case (2) above, if $Y(u) \neq Y(B_u)$, then $Last(u)$ can be found as the origin of the tail of $\sigma(e_1)$.

By Lemma 17(a) and (1) and (2) above, we can decide for all $u \in V_{\mathcal{B}}$ whether $f(u) = \emptyset$. It is then easy to compute $pre_{f \neq \emptyset}(u)$ for all $u \in V_{\mathcal{B}}^+$ by means of pointer doubling. Finally, we use parts (b) and (c) of Lemma 17 and the above observation to determine all remaining values of f. □

We would like to compute X by means of pointer doubling applied to $First \circ f$ and $Last \circ f$. One complication preventing us from doing this in a straightforward way is the possibility of vertices $u \in V_{\mathcal{B}}$ with $f(u) = \emptyset$. We get around this difficulty as follows.

For all $u \in V_{\mathcal{B}}$, introduce a new vertex $\bar{u}$ and put $\overline{suc}(u) = \bar{u}$ and $\overline{suc}(\bar{u}) = suc(u)$. For all $U \subseteq V_{\mathcal{B}}$, let $\hat{U} = U \cup \{\bar{u} \mid u \in U\}$. Let $\overline{G_{\mathcal{B}}}$ be the directed graph $(\widehat{V_{\mathcal{B}}}, \{(u, \overline{suc}(u)) \mid u \in \widehat{V_{\mathcal{B}}}\})$, and let $\mathcal{S}$ be the set of nonempty segments of cycles of $\overline{G_{\mathcal{B}}}$. In the following, the symbols $\overline{First}$, $\overline{Last}$, and $\overline{[\cdot, \cdot]}$ will have the meaning of the corresponding symbols without bars, but defined relative to the cycles of $\overline{G_{\mathcal{B}}}$. Informally, we have introduced a new dummy vertex $\bar{u}$ "on" each black edge $(u, suc(u))$. Correspondingly, define the level of $\bar{u}$ to be the level of u, for all $u \in V_{\mathcal{B}}$.

DEFINITION. For all $u \in V_{\mathcal{B}}$, let $\bar{f}(u)$ and $\bar{f}(\bar{u})$ be given as follows:

(1) If $f(u) \neq \emptyset$, let $\bar{f}(u) = \overline{[First(f(u)), Last(f(u))]}$ and $\bar{f}(\bar{u}) = \{\overline{Last(f(u))}\}$.

(2) If $f(u) = \emptyset$, let $\bar{f}(u) = \bar{f}(\bar{u}) = \{\overline{Last(f(pre_{f \neq \emptyset}(u)))}\}$.

Here $First(f(u))$ and $Last(f(u))$, for all $u \in V_{\mathcal{B}}$ with $f(u) = \mathcal{V}(p_{\mathcal{B}}(B_u))$, are to be understood as arbitrary but fixed elements of $V_{\mathcal{B}}$ with $f(u) = [First(f(u)), Last(f(u))]$.

Clearly, the values of $\bar{f}$ on single vertices can be computed from those of f in $O(\log n)$ time with $O(n/\log n)$ processors. Define $\overline{p_{\mathcal{B}}}(I)$, for all $I \in \mathcal{I}$, to be the cycle of $\overline{G_{\mathcal{B}}}$ containing $\bar{f}(I)$. The property stated in the following lemma is the reason for introducing $\bar{f}$. Informally, the lemma says that given a diagram as that of Fig. 17, $\bar{f}^{(k)}(I)$ may be computed by following upward for k steps the leftmost and the rightmost path starting at vertices in I, and then taking whatever lies between the end vertices reached. The only exceptional case occurs when the two paths end in the same vertex.

LEMMA 19. *Let* $k \geqq 1$, $I = \overline{[u, v]} \in \mathcal{I}$ *and*

$$J = \overline{[(\overline{First} \circ \bar{f})^{(k)}(u), (\overline{Last} \circ \bar{f})^{(k)}(v)]}.$$

Then

(a) *If* $J \cap V \neq \emptyset$, *then* $\bar{f}^{(k)}(I) = J$.

(b) *If* $J \cap V = \emptyset$, *then either* $\bar{f}^{(k)}(I) = J$, *or else* $\bar{f}^{(k)}(I) = \mathcal{V}(\overline{p_{\mathcal{B}}}^{(k)}(I))$.

Proof. For $k = 1$ this follows from the properties of f, as expressed in Lemma 17 (see Fig. 17). The general claim may be verified by induction on k. □

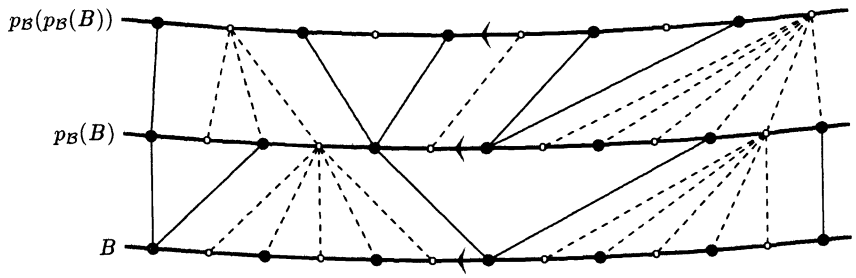


FIG. 17. *A pictorial representation of an example function* $\bar{f}$. *For each vertex* u, $\bar{f}(u)$ *is indicated by lines from* u *to* $\overline{First}(\bar{f}(u))$ *and* $\overline{Last}(\bar{f}(u))$. *Fully drawn lines are used when* $u \in V_{\mathcal{B}}$ *and* $f(u) \neq \emptyset$, *dashed lines otherwise. Dummy vertices are shown smaller and white.*

By Lemma 19, $\bar{f}^{(k)}(u) \in \mathcal{I}$ for all $u \in \widehat{V_{\mathcal{B}}}$ and all $k \geqq 1$. Hence "knowing" $\bar{f}^{(k)}(u)$ amounts to knowing two vertices $v, w \in \widehat{V_{\mathcal{B}}}$ with $\bar{f}^{(k)}(u) = \overline{[v, w]}$.

LEMMA 20. *Let* $j, k \geqq 1$ *and* $U \subseteq \widehat{V_{\mathcal{B}}^+}$. *Suppose that for all* $u \in U$, $\bar{f}^{(j)}(u)$ *is known, and* $\bar{f}^{(k)}(v)$ *is known for all* $v \in \mathcal{V}(\overline{p_{\mathcal{B}}}^{(j)}(\{u\}))$. *Then, if a processor is associated with each vertex in* $\widehat{V_{\mathcal{B}}^+}$, $\bar{f}^{(j+k)}(u)$ *can be computed in constant time for all* $u \in U$.

Proof. Since $\bar{f}^{(j+k)}(u) = \bar{f}^{(k)}(\bar{f}^{(j)}(u))$ for all $u \in U$, we may simply use Lemma 19 with $I = \bar{f}^{(j)}(u)$. To decide whether $\bar{f}^{(k)}(I) \cap V = \emptyset$ or $\bar{f}^{(k)}(I) = \mathcal{V}(\overline{p_{\mathcal{B}}}^{(k)}(I))$ in case (b) of Lemma 19, it suffices to choose one vertex v on the cycle of $\overline{G_{\mathcal{B}}}$ containing I with $\bar{f}^{(k)}(v) \cap V \neq \emptyset$, and to test whether $v \in I$. Since $\bar{f}^{(k)}(u) = \{u\}$ for $u \in \widehat{\mathcal{V}(B_0)}$, the test need only be carried out if $I \subseteq \widehat{V_{\mathcal{B}}^+}$. □

LEMMA 21. *X can be computed in* $O(\log n)$ *time with* $O(n/\log n)$ *processors.*

Proof. By induction on k it follows trivially that for all $u \in V_{\mathcal{B}}$ and all $k \geqq 1$, $f^{(k)}(u) = \bar{f}^{(k)}(u) \cap V$, so that for all $u \in V_{\mathcal{B}}$, $u \in X$ if and only if $r \in \bar{f}^{(n)}(u)$. Hence the problem reduces to that of computing $\bar{f}^{(n)}(u)$ for all $u \in V_{\mathcal{B}}$. Note first that this is easy if n processors are available: Simply use Lemma 20 repeatedly with $U = \widehat{V_{\mathcal{B}}^+}$ and with $j = k$ equal to $1, 2, 4, \cdots, 2^{\lfloor \log n \rfloor}$. To show that the same can be achieved with $O(n/\log n)$ processors, we describe the construction of a smaller instance of the same

problem. We proceed as in the proof of Lemma 13, except that we must be slightly more careful regarding the choice of an independent set.

Partition $\widehat{V_{\mathscr{B}}^+}$ into the sets V_{odd} and V_{even} of vertices whose levels are odd and even, respectively. If $|V_{\text{odd}}| > |V_{\text{even}}|$, let $S = V_{\text{odd}}$, else let $S = V_{\text{even}}$. By Lemma 12, S can be identified in $O(\log n/\log\log n)$ time with $O(n \log\log n/\log n)$ processors, provided that the level of each vertex in $\widehat{V_{\mathscr{B}}}$ is known. Now for all $u \in \widehat{V_{\mathscr{B}}^+}\backslash S$, compute $\bar{f}^{(2)}(u)$ by Lemma 20. Sparing the reader the formal definition of the relevant class of problem instances, we state that by replacing $\widehat{V_{\mathscr{B}}}$ by $\widehat{V_{\mathscr{B}}}\backslash S$, $\bar{f}$ by the restriction of $\bar{f}^{(2)}$ to $\widehat{V_{\mathscr{B}}}\backslash S$, etc., we can use another parallel prefix computation to construct a problem instance with the same essential characteristics as the original instance (in particular, and most importantly, Lemma 19 holds with $\bar{f}$ replaced by $\bar{f}^{(2)}$) and of at most half its size. Here we define the size of an instance as the number of vertices of positive level. This is legitimate since Lemma 20 does not require processors to be associated with vertices of level zero. Once the smaller problem has been solved, we know $\bar{f}^{(n)}(u)$ for all $u \in \widehat{V_{\mathscr{B}}}\backslash S$, and we can determine $\bar{f}^{(n)}(u)$ for all $u \in S$ by Lemma 20 with $U = S$ and $j = 1$, $k = n$. Hence we may finally appeal to Lemma 1 of [12] and conclude that $\bar{f}^{(n)}(u)$ can be computed for all $u \in V_{\mathscr{B}}$ in $O(\log n)$ time with $O(n/\log n)$ processors. □

THEOREM 2. *Suppose that time $T(n) \geqq \log n$ and $p(n)$ processors suffice to construct a planar embedding of a planar graph on n vertices and to compute a* BFS *tree with a given root of a planar graph on $3n$ vertices. Then, given an undirected connected planar graph $G = (V, E)$ on n vertices and a vertex $r \in V$, a* DFS *tree of G rooted at r can be computed in time $O(T(n))$ by a* PRIORITY PRAM *with $p(n)$ processors.*

Proof. Consider the algorithm given in the proof of Theorem 1. If G is biconnected, the distance computation in step (5) can alternatively be done by constructing a BFS tree T of $G'_{\mathscr{D}}$ rooted at F_0 and computing, by means of pointer doubling, the depth in T of each $F \in \mathscr{F}$. The same method works if G is not biconnected, except that one must first construct an auxiliary graph as in the proof of Lemma 14(b). Since steps that can be executed optimally in time $O(\log n)$ certainly can be executed optimally in time $O(T(n))$, it now suffices to show how to carry out step (14) in $O(\log n)$ time with $O(n/\log n)$ processors. By Lemma 21, X can be computed within the stated time and processor bounds. Hence all that remains is to determine the L-successor of $X(p_{\mathscr{B}}(L))$, for all $L \in \mathscr{L}^+$. For all $u \in V_{\mathscr{B}}$, label u with the distance on $\overrightarrow{B_u}$ from u to $X(B_u)$. Then label each selected representative with the label of its origin. Now for each $L \in \mathscr{L}^+$, the L-successor of $X(p_{\mathscr{B}}(L))$ is the selected representative in L with maximum label which, by Lemma 14(e), can be identified within the stated resource bounds. □

Remark. A straightforward implementation of the algorithm implicit in Theorem 2, exclusive of subroutines for constructing BFS trees and planar embeddings, uses space which is polynomial in n. By the result of [11], the space requirements can be reduced to $O(n^{1+\varepsilon})$, for any fixed $\varepsilon > 0$.

REFERENCES

[1] A. AGGARWAL AND R. J. ANDERSON, *A random $\mathcal{NC}$ algorithm for depth first search*, Combinatorica, 8 (1988), pp. 1-12.

[2] A. AGGARWAL, R. J. ANDERSON, AND M. Y. KAO, *Parallel depth-first search in general directed graphs*, Tech. Report 264, Indiana University, Bloomington, IN, 1988.

[3] R. COLE AND U. VISHKIN, *Faster optimal parallel prefix sums and list ranking*, Inform. and Comput., 81 (1989), pp. 334-352.

[4] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Comput., 64 (1985), pp. 2-22.

[5] D. EPPSTEIN AND Z. GALIL, *Parallel algorithmic techniques for combinatorial computation*, Ann. Rev. Comput. Sci., 3 (1988), pp. 233–283.
[6] S. EVEN, *Graph Algorithms*, Pitman, London, 1979.
[7] A. GIBBONS AND W. RYTTER, *Efficient Parallel Algorithms*, Cambridge University Press, Cambridge, 1988.
[8] A. V. GOLDBERG, S. A. PLOTKIN, AND G. E. SHANNON, *Parallel symmetry-breaking in sparse graphs*, SIAM J. Discrete Math., 1 (1988), pp. 434–446.
[9] A. V. GOLDBERG, S. A. PLOTKIN, AND P. M. VAIDYA, *Sublinear-time parallel algorithms for matching and related problems*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 174–185.
[10] A. GORALČÍKOVÁ AND V. KOUBEK, *Some improved parallelisms for graphs*, in Proc. 12th Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, 233, J. Gruska, B. Rovan, and J. Wiedermann, eds., Springer-Verlag, Berlin, New York, 1986, pp. 379–385.
[11] T. HAGERUP, *On saving space in parallel computation*, Inform. Process. Lett., 29 (1988), pp. 327–329.
[12] ———, *Optimal parallel algorithms on planar graphs*, Inform. and Comput., 84 (1990), pp. 71–96.
[13] M. Y. KAO, *All graphs have cycle separators and planar directed depth-first search is in* DNC, in Proc. 3rd Aegean Workshop on Computing, Lecture Notes in Computer Science, 319, J. H. Reif, ed., Springer-Verlag, Berlin, New York, 1988, pp. 53–63.
[14] V. RAMACHANDRAN AND J. REIF, *An optimal parallel algorithm for graph planarity*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Los Alamitos, CA, 1989, pp. 282–287.
[15] J. H. REIF, *Depth-first search is inherently sequential*, Inform. Process. Lett., 20 (1985), pp. 229–234.
[16] G. E. SHANNON, *A linear-processor algorithm for depth-first search in planar graphs*, Inform. Process. Lett., 29 (1988), pp. 119–123.
[17] J. R. SMITH, *Parallel algorithms for depth-first searches* I. *Planar graphs*, SIAM J. Comput., 15 (1986), pp. 814–830.
[18] R. TARJAN, *Depth-first search and linear graph algorithms*, SIAM J. Comput., 1 (1972), pp. 146–160.
[19] R. E. TARJAN AND U. VISHKIN, *An efficient parallel biconnectivity algorithm*, SIAM J. Comput., 14 (1985), pp. 862–874.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 705–710, August 1990

009

A NOTE ON OPTIMAL BIN PACKING AND OPTIMAL BIN COVERING WITH ITEMS OF RANDOM SIZE*

WANSOO T. RHEE†

Abstract. Consider a probability measure μ on $[0, 1]$ and independent identically distributed random variables $X_1, \cdots, X_n$ distributed according to μ. Denote by $Q_n = Q_n(X_1, \cdots, X_n)$ the minimum number of unit-size bins needed to pack items of size $X_1, \cdots, X_n$. Previous estimates of Q_n are considerably improved and simplified. Similar estimates are obtained for the maximum number of unit-size bins that can be covered by $X_1, \cdots, X_n$.

Key words. bin packing, bin covering, stochastic

AMS(MOS) subject classifications. 90B99, 60K30

1. Introduction and results. The celebrated bin packing problem requires finding the minimum number of unit-size bins $Q_n(X_1, \cdots, X_n)$ needed to pack a given collection $X_1, \cdots, X_n$ of items with sizes in $[0, 1]$ subject to the requirement that the sum of the sizes of the items allocated to any bin does not exceed one. Its companion problem, the bin-covering problem, requires finding the maximum number $Q'_n(X_1, \cdots, X_n)$ of unit-size bins that can be covered by a given collection $X_1, \cdots, X_n$ of items with sizes $[0, 1]$ subject to the requirement that the sum of the sizes of the items allocated to any bin is greater than or equal to one.

In this paper, we consider the same stochastic model as in our previous work [2]–[5]. We consider a probability measure μ on $[0, 1]$, on which no regularity assumptions are made. We consider n items $X_1, \cdots, X_n$ independent identically distributed (i.i.d.) according to μ. (For simplicity, we denote by X_k both item names and item sizes.) We define $c(\mu) = \lim_{n\to\infty} E(Q_n)/n$, $d(\mu) = \lim_{n\to\infty} E(Q'_n)/n$. The purpose of this paper is to strengthen and streamline previous results of Rhee and Talagrand. Before we discuss the relationship between the results of the present paper and the previous ones, we recall some basic notation.

For $k \geqq 1$, we set

$$R_k = \left\{ (x_1, \cdots, x_k) \in \mathbf{R}^k;\ 0 \leqq x_1 \leqq \cdots \leqq x_k \leqq 1,\ \sum_{i \leqq k} x_i \leqq 1 \right\}$$

and

$$S_k = \left\{ (x_1, \cdots, x_k) \in \mathbf{R}^k;\ 0 \leqq x_1 \leqq \cdots \leqq x_k \leqq 1,\ 1 \leqq \sum_{i \leqq k} x_i \leqq 3 \right\}.$$

Both are compact metric spaces. For a compact metric space S we denote by $M_1(S)$ the set of probability measures on S. For $x \in [0, 1]$, we denote by δ_x the unit mass concentrated at x, i.e., for a Borel set G we have $\delta_x(G) = 1$ if x belongs to G and $\delta_x(G) = 0$ otherwise.

For $\nu \in M_1(T_k)$, where $T_k = R_k$ or S_k, we consider the measure $\mathcal{P}_k(\nu) \in M_1([0, 1])$ given, for each Borel set G of $[0, 1]$, by

$$\mathcal{P}_k(\nu)(G) = \int_{T_k} \frac{1}{k} \sum_{i \leqq k} \delta_{x_i}(G)\, d\nu(x_1, \cdots, x_k).$$

* Received by the editors December 9, 1988; accepted for publication (in revised form) November 7, 1989. This research was supported in part by National Science Foundation grant CCR-8801517.

† The Ohio State University, Faculty of Management Science, Columbus, Ohio 43210.

To motivate our results, and make the paper more self-contained, we recall the following.

THEOREM 1. *Given a probability μ on $[0, 1]$, the following hold.*

(1) *There exists a nonnegative sequence $(\alpha_k)_{k\geqq 0}$ with $\sum_{k\geqq 0}\alpha_k = 1$, and a sequence $\nu_k \in M_1(R_k)$ such that*

$$\mu = \alpha_0\delta_0 + \sum_{k\geqq 1}\alpha_k\mathscr{P}_k(\nu_k) \tag{1}$$

and $\sum_{k\geqq 1}\alpha_k/k \leqq c(\mu)$.

(2) *There exists a nonnegative sequence $(\alpha_k)_{k\geqq 0}$ with $\sum_{k\geqq 0}\alpha_k = 1$, and a sequence $\nu_k \in M_1(S_k)$ such that (1) holds, and $\sum_{k\geqq 1}\alpha_k/k \geqq d(\mu)$.*

The first part of Theorem 1 is obtained in [2], by a compactness argument. The second part is proved along the same lines in [5]. Given a decomposition of μ as in (1), it is proved in [2] that $c(\mu) \leqq \sum_{k\geqq 1}\alpha_k/k$. Much of [3], and part of [4] are devoted to make this statement quantitative, and to give estimates of the type $Q_n \leqq n\sum_{k\geqq 1}\alpha_k/k + \textit{error term}$. All these efforts are subsumed by the next result, which replaces error terms that are powers of n by a power of $\log n$.

THEOREM 2. *Consider a nonnegative sequence $(\alpha_k)_{k\geqq 0}$ with $\sum_{k\geqq 0}\alpha_k = 1$, and a sequence $\nu_k \in M_1(R_k)$. Consider the measure*

$$\mu = \alpha_0\delta_0 + \sum_{k\geqq 1}\alpha_k\mathscr{P}_k(\nu_k).$$

Consider a sequence $s_1, \cdots, s_n$ of items $(n \geqq 2)$. Let

$$D = D(s_1, \cdots, s_n) = \sup_{0\leqq t\leqq 1}\{\text{card}\,\{i \leqq n, s_i \geqq t\} - n\mu([t, 1])\}$$

so $D \geqq 0$. Then

$$Q_n(s_1, \cdots, s_n) \leqq D + n\sum_{k\geqq 1}\frac{\alpha_k}{k} + K(\log n)^2 \tag{2}$$

where K is a universal constant (i.e., independent of μ, n, and $s_1, \cdots, s_n$).

Given a decomposition of the type (1), estimates of the type $Q'_n \geqq n\sum_{k\geqq 1}\alpha_k/k$-*error term* are proved in [5]. The error term of [5] is not $o(n^{-1/2})$. This creates complications in proving the central limit theorem of [5] for Q'_n, and forces to put some (mild) restrictions on the sequence $(\alpha_k)_{k\geqq 1}$. The much improved estimate (3) below for Q'_n allows us to remove all these restrictions and complications and to prove the central limit theorem for Q'_n of [5] along the same lines as the theorem for Q_n of [4], by just "reversing inequalities."

THEOREM 3. *Consider a nonnegative sequence $(\alpha_k)_{k\geqq 0}$ with $\sum_{k\geqq 0}\alpha_k = 1$, and a sequence $\nu_k \in M_1(S_k)$. Consider the measure*

$$\mu = \alpha_0\delta_0 + \sum_{k\geqq 1}\alpha_k\mathscr{P}_k(\nu_k).$$

Consider a sequence $s_1, \cdots, s_n$ of items $(n \geqq 2)$. Let

$$D' = D'(s_1, \cdots, s_n) = \sup_{0\leqq t\leqq 1}\{n\mu([t, 1]) - \text{card}\,\{i \leqq n; s_i \geqq t\}\}.$$

Thus $D' \geqq 0$. Then

$$Q'_n(s_1, \cdots, s_n) \geqq n\sum_{k\geqq 1}\frac{\alpha_k}{k} - D' - K(\log n)^2. \tag{3}$$

It is interesting and possibly challenging to determine whether the error terms in (2) and (3) can be improved.

2. Proofs. The proofs will make essential use of the following observation, usually called the Carathéodory theorem. If a point $z \in \mathbf{R}^r$ is a convex combination of a family of points, it is a convex combination of at most $(r+1)$ of these points. We denote by $\mathcal{G}$ the family of functions of the type

$$g(t) = \text{card}\,\{i \leqq m;\ s_i \geqq t\}$$

where m is arbitrary and $0 \leqq s_i \leqq 1$, $\sum_{i \leqq m} s_i \leqq 1$.

We first consider Theorem 2.

LEMMA 1. *Consider $k \geqq 1$, $p \geqq 4$ such that $2^{p-3} \geqq k$, and an index set L with* card $L \geqq 2^p$. *For $l \in L$, consider $0 \leqq \beta_l < 1$, and $g_l \in \mathcal{G}$. then there exists a set $L' \subset L$ with* card $L' \leqq 2^{p-1}$, *and for $l \in L'$ numbers $0 \leqq \gamma_l < 1$, integers n_l such that*

$$\sum_{l \in L'} (\gamma_l + n_l) \leqq \sum_{l \in L} \beta_l + 4k, \tag{4}$$

$$\min\left(2^k, \sum_{l \in L} \beta_l g_l(t)\right) \leqq \sum_{l \in L'} (\gamma_l + n_l) g_l(t) + 1. \tag{5}$$

Proof. We set $g = \sum_{l \in L} \beta_l g_l$ and we observe that g is left continuous. Set $\alpha = 1 + 4k2^{-p}$. We define by induction a sequence u_i as follows. We set

$$u_1 = \sup\,\{t;\ g(t) \geqq 1\}$$

so that $g(u_1) \geqq 1$. We then set

$$u_{i+1} = \sup\,\{t;\ g(t) \geqq \alpha g(u_i)\}$$

so that $g(u_{i+1}) \geqq \alpha g(u_i)$. The construction stops at the first u_r for which either $g(u_r) > 2^k$ or $g(t) < \alpha g(u_r)$ for $0 \leqq t < u_r$. We have

$$2^k \geqq g(u_{r-1}) \geqq \alpha^{r-2} g(u_1) \geqq \alpha^{r-2}.$$

Since $8k2^{-p} \leqq 1$, we have $\alpha \geqq 2^{4k2^{-p}}$, and thus $(r-s)4k2^{-p} \leqq k$, so that

$$r \leqq 2 + 2^{p-2} < 2^{p-1}.$$

We now apply the Carathéodory theorem to the convex combination $(\sum_{l \in L} \beta_l)^{-1} \sum_{l \in L} \beta_l g_l$. We can find $L' \subset L$ with card $L' \leqq r+1 \leqq 2^{p-1}$, and for $l \in L'$ numbers $\xi_l \geqq 0$ such that $\sum_{l \in L'} \xi_l = \sum_{l \in L} \beta_l$ and that

$$\forall i, \quad 1 \leqq i \leqq r, \quad \sum_{l \in L'} \xi_l g_l(u_i) = \sum_{l \in L} \beta_l g_l(u_i) = g(u_i).$$

We set $n_l = \lfloor \alpha \xi_l \rfloor$, $\gamma_n = \alpha \xi_l - n_l$. We observe that

$$\sum_{l \in L'} (n_l + \xi_l) = \alpha \sum_{l \in L} \beta_l \leqq \sum_{l \in L} \beta_l + 4k2^{-p} \cdot 2^p,$$

which proves (5).

Consider $t \in [0, 1]$. If $t > u_1$, then $g(t) < 1$, so that (6) holds. If $t < u_r$, and $g(u_r) > 2^k$, we have

$$\sum_{l \in L'} (\gamma_l + n_l) g_l(t) = \alpha \sum_{l \in L'} \xi_l g_l(t) \geqq \sum_{l \in L'} \xi_l g_l(u_r) = g(u_r) > 2^k,$$

so that (6) holds. In the other cases, there is a largest $i \leqq r$ such that $t < u_i$ and $g(t) \leqq \alpha g(u_i)$, by definition of u_{i+1}. We have

$$g(t) \leqq \alpha g(u_i) = \sum_{l \in L'} (\gamma_l + n_l) g_l(u_i) \leqq \sum_{l \in L'} (\gamma_l + \eta_l) g_l(t).$$

This completes the proof.

Proof of Theorem 1. By (1) and the definition of D, we observe that for all $t>0$,

$$f(t)=\text{card}\,\{i\leqq n;\ s_i\geqq t\}$$

$$\leqq D+n\sum_{k\geqq 1}\frac{\alpha_k}{k}\int_{R_k}\text{card}\,\{i\leqq k;\ x_i\geqq t\}\,d\nu_k(x_1,\cdots,x_k).$$

Thus, approximating the integrals by finite averages, we can find a finite index set L, for $l\in L$ a function $g_l\in\mathcal{G}$ and numbers ξ_l with $\sum_{l\in L}\xi_l\leqq n\sum_{i\geqq 1}\alpha_i/i$ such that

$$\forall j\leqq n,\quad \text{card}\,\{i\leqq n;\ s_i\geqq s_j\}\leqq D+1+\sum_{l\in L}\xi_l g_l(s_j). \tag{6}$$

Let k be the smallest integer with $n<2^k$. Using the Carathéodory theorem, we can assume that card $L\leqq n+1\leqq 2^k$.

Now it is clear that (6) implies

$$\forall t\geqq 0,\quad \text{card}\,\{i\leqq n;\ s_i\geqq t\}\leqq D+1+\sum_{l\in L}\xi_l g_l(t). \tag{7}$$

Let p_0 be the smallest integer for which $2^{p_0}\geqq 8k$. Since $n\geqq 2$, we have that $k\geqq 2$ and $p_0\geqq 4$. Also $2^{p_0-1}<8k$. Let $\beta_l=\xi_l-\lfloor\xi_l\rfloor$. We observe that (6) implies

$$\min\left(2^k,\sum_{l\in L}\beta_l g_l(t)\right)\leqq 1+\sum_{l\in L'}n_l g_l(t)+\min\left(2^k,\sum_{l\in L'}\gamma_l g_l(t)\right).$$

So, we can iteratively apply Lemma 1 for the values $p=k,\cdots,p_0$. We sum the corresponding inequalities. We thus find integers n_l, $l\in L$, a set $L'\subset L$ with card $L'\leqq 2^{p_0-1}<8k$, and for $l\in L'$ a number $0\leqq\beta_l'\leqq 1$ such that

$$\sum_{l\in L}n_l+\sum_{l\in L'}\beta_l'\leqq\sum_{l\in L}\beta_l+4k(k-3)$$

and

$$\min\left(2^k,\sum_{l\in L}\beta_l g_l(t)\right)\leqq\sum_{l\in L}n_l g_l(t)+\sum_{l\in L'}\beta_l' g_l(t)+k-3.$$

If we combine with (8), we see that we have found integers $m_l(=n_l+\lfloor\xi_l\rfloor)$ such that

$$\forall t\geqq 0,\quad \text{card}\,\{i\leqq n;\ s_i\geqq t\}\leqq D+k-2+\sum_{l\in L'}\beta_l' g_l(t)+\sum_{l\in L}m_l g_l(t)$$

and that

$$\sum_{l\in L}m_l\leqq n\sum_{i\geqq 1}\frac{\alpha_i}{i}+4k(k-3).$$

We now appeal to Lemma 2 of [3] to see that our family of items can be split into three families H_1, H_2, and H_3 such that

$$\forall t\geqq 0,\quad \text{card}\,\{s\in H_1;\ s\geqq t\}\leqq\sum_{l\in L}m_l g_l(t), \tag{8}$$

$$\forall t\geqq 0,\quad \text{card}\,\{s\in H_2;\ s\geqq t\}\leqq\sum_{l\in L'}\beta_l' g_l(t), \tag{9}$$

$$\text{card}\,H_3\leqq D+k-1. \tag{10}$$

It is clear that H_1 can be packed in at most $\sum_{l\in L}m_l$ bins. The sum of the sizes of the elements of H_2 is given by

$$\int_0^1\text{card}\,\{s\in H_2:\ s\geqq t\}\,dt.$$

Since $\beta_l' \leqq 1$, and $\int_0^1 h(t)\,dt \leqq 1$ for $h \in \mathcal{G}$, we see from (10) that the sum of the sizes of the items in H_2 is $\leqq \text{card } L' \leqq 8k$. Thus these items can be packed in at most $16k$ bins. Since H_3 can be packed in at most $D+k-1$ bins, we have succeeded in using at most

$$h \sum_{i \geqq 1} \frac{\alpha_i}{i} + 4k(k-3) + 16k + D + k - 1$$

bins. The proof is complete. □

We now consider Theorem 3. We denote by $\mathcal{G}$ the family of functions of the type

$$g(t) = \text{card}\,\{i \leqq m;\ s_i \geqq t\}$$

where m is arbitrary and $0 \leqq s_i \leqq 1$, $\sum_{i \leqq m} s_i \geqq 1$. The proof of Theorem 3 is very similar to the proof of Theorem 2, so we only indicate the necessary modifications. For a function g, we set $g^+(t) = \lim_{u \to t^+} g(u)$. The following will replace Lemma 1.

LEMMA 2. *Consider $k \geqq 1$, $p \geqq 4$, with $2^{p-3} \geqq k$ and an index set L with* card $L \leqq 2^p$. *For $l \in L$, consider $0 \leqq \beta_l < 1$, and $g_l \in \mathcal{G}$. Set $g = \sum_{l \in L} \beta_l g_l$. Assume that $g^+(0) \leqq 2^k$. Then there exists a set $L' \subset L$ with* card $L' \leqq 2^{p-1}$ *and for $l \in L'$ numbers $0 \leqq \gamma_l < 1$, integers n_l such that*

$$\sum_{l \in L'} (\gamma_l + n_l) \geqq \sum_{l \in L} \beta_l - 4k, \tag{11}$$

$$\forall t \in]0, 1], \quad \sum_{l \in L'} (\gamma_l + n_l) g_l(t) \leqq 1 + g(t), \tag{12}$$

$$\sum_{l \in L'} \gamma_l g_l^+(0) \leqq 2^k. \tag{13}$$

Proof. We set $\alpha = 1/(1 - k2^{-p+2}) \geqq \exp k2^{-p+2} \geqq 2^{k2^{-p+2}}$. We construct the sequence u_i as in Lemma 1, but this time we stop only when we construct $u_r > 0$ such that

$$t < u_r \Rightarrow g(t) < \alpha g(u_r).$$

Since $g(u_r) < g^+(0) \leqq 2^k$, we again have $r \leqq 1 + 2^{p-2}$. We set $u_{r+1} = 0$. We observe that $g^+(u_{i+1}) \leqq \alpha g(u_i)$, and $g^+(u_1) \leqq 1$.

There exists $\varepsilon > 0$ such that all the functions g_l, $l \in L$ are constant in all the intervals $]u_i, u_i + \varepsilon]$, $1 \leqq i \leqq r+1$. Using the Carathéodory theorem, we can find $L' \subset L$ with card $L' \leqq r + 2 \leqq 2^{p-1}$ and for $l \in L'$ numbers $0 \leqq \xi_l \leqq 1$ such that

$$\forall i, \quad 1 \leqq i \leqq r+1, \quad \sum_{l \in L'} \xi_l g_l(u_i + \varepsilon) = g(u_i + \varepsilon).$$

We set $n_l = \lfloor \xi_l / \sigma \rfloor$, $\gamma_l = \eta_l / \alpha - n_l$. Given $t \in [0, 1]$, consider the largest i such that $t \geqq u_i$. Since, for all l, we have $g_l(t) \leqq g_l^+(u_i) = g_l(u_i + \varepsilon)$, we have

$$\sum_{l \in L'} (n_l + \gamma_l) g_l(t) \leqq \frac{1}{\alpha} \sum_{l \in L'} \xi_l g_l(u_i + \varepsilon) = \frac{1}{\alpha} g(u_i + \varepsilon) = \frac{1}{\alpha} g^+(u_i).$$

If $i = 1$, then $g^+(u_i) \leqq 1$, so that (13) holds since $\alpha \geqq 1$. Otherwise, $g^+(u_i) \leqq \alpha g(u_{i-1}) \leqq \alpha g(t)$, and (13) holds again. To complete the proof, observe that

$$\sum_{l \in L'} \gamma_l g_l^+(0) \leqq \sum_{l \in L'} (\gamma_l + n_l) g_l(\varepsilon) = \frac{1}{\alpha} g(\varepsilon) \leqq g^+(0)$$

and that

$$\sum_{l \in L'} (n_l + \gamma_l) = \frac{1}{\alpha} \sum_{l \in L'} \xi_l = \frac{1}{\alpha} \sum_{l \in L} \beta_l \geqq \sum_{l \in L} \beta_l - 2^p \left(1 - \frac{1}{\alpha}\right) \geqq \sum_{l \in L} \beta_l - 4k. \qquad \square$$

To prove Theorem 3, we proceed very much as in the case of Theorem 2. Let k be the smallest integer such that $2^k \geqq 2n+2 \geqq n+D+2$. First we find a set L with card $L \leqq n+2 \leqq 2^k$ and for $l \in L$, we find $\xi_l \geqq 0$, $g_l \in \mathcal{G}$ such that

$$\sum_{l \in L} \xi_l \geqq n \sum_{k \geqq 1} \frac{\alpha_k}{k} - 1,$$

$$n+D+1 \geqq \sum_{l \in L} \xi_l g_l(0),$$

$$\forall j \leqq n, \quad \text{card}\,\{i \leqq n;\ s_i \geqq s_j\} + D + 1 \geqq \sum_{l \in L} \xi_l g_l(s_i).$$

This last condition is easily seen to imply that

$$\forall t \geqq 0, \quad \text{card}\,\{i \leqq n;\ s_i \geqq t\} + D + 1 \geqq \sum_{l \in L} \xi_l g_l(t).$$

we set $\beta_l = \xi_l - \lfloor \xi_l \rfloor$, and we apply Lemma 2 to $\sum_{l \in L} \beta_l g_l$. Letting p_0 be the smallest integer for which $2^{p_0-3} \geqq k$, we reiterate the use of Lemma 2 for $k, \cdots, p_0$, to obtain integers $(n_l)_{l \in L}$ such that

$$\sum_{l \in L} n_l \geqq n \sum_{i \geqq 1} \frac{\alpha_i}{i} - 4(k-p_0+1)k - 2^{p_0-1}$$

and

$$\forall t \in [0,1], \quad \text{card}\,\{i \leqq n;\ s_i \geqq t\} + D + (k-p_0+2) \geqq \sum_{l \in L} n_l g_l(t).$$

It is easily seen that this latter condition implies that we can cover at least $\sum_{l \in L} n_l - D - (k-p_0+2)$ bins with $s_1, \cdots, s_n$. This implies the result.

It was mentioned in [5] that bin covering seems more difficult than bin packing. This statement appears to be incorrect when we use the present approach. Implicit in the proof of Theorem 2 is the possibility of suitably choosing a few bins and using their items to cover many of the small gaps that may remain in other bins. This possibility was not used in the strategy described in the proof of Theorem 3 of [5], contributing to the inefficiency of the bound there.

REFERENCES

[1] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Approximation algorithms for bin packing—an updated survey*, in Algorithm Design for Computer System Design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, Berlin, New York, 1984, pp. 49-106.

[2] W. RHEE, *Optimal bin packing with items of random sizes*, Math. Oper. Res., 13 (1988), pp. 140-151.

[3] W. RHEE AND M. TALAGRAND, *Optimal bin packing with items of random size*—II, SIAM J. Comput., 18 (1989), pp. 139-151.

[4] ———, *Optimal bin packing with items of random size*—III, SIAM J. Comput., 18 (1989), pp. 473-486.

[5] ———, *Optimal bin covering with items of random size*, SIAM J. Comput., 18 (1989), pp. 487-498.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 711-717, August 1990

010

SOME OBSERVATIONS ON PARALLEL ALGORITHMS FOR FAST EXPONENTIATION IN GF(2^n)*

D. R. STINSON†

Abstract. A normal basis representation of GF(2^n) allows squaring to be accomplished by a cyclic shift. Algorithms for multiplication in GF(2^n) using a normal basis have been studied by several researchers. In this paper, algorithms for performing exponentiation in GF(2^n) using a normal basis, and how they can be speeded up by using parallelization, are investigated.

Key words. finite field, normal basis, exponentiation, parallel algorithm

AMS(MOS) subject classifications. 68Q25, 12E20

1. Introduction. The problem of exponentiating in GF(2^n) has many important applications in error-correcting codes and in cryptography. In cryptographic applications especially, it is often necessary to use very large fields. For example, suppose we wish to employ the Diffie-Hellman key-exchange protocol in GF(2^n) (see [5]). In order that an opponent not be able to compute discrete logarithms in GF(2^n), it is preferable to use a field GF(2^n) where $n > 800$ (see [13]). Consequently, it is crucial to have algorithms for arithmetic in GF(2^n) that are practical for large values of n.

This paper is an investigation of normal basis algorithms for fast exponentiation in GF(2^n), and how they can be speeded up by using parallelization. We use a divide-and-conquer technique due to Agnew et al. described in [1] and [2].

The Galois field GF(2^n) is an n-dimensional vector space over GF(2). A basis for GF(2^n) of the form $\{\beta, \beta^2, \beta^4, \cdots, \beta^{2^{n-1}}\}$ is called a *normal basis*. The element β is the generator of the normal basis. It is well known that GF(2^n) has a normal basis for every $n \geqq 1$ (see, for example, [9]). If field elements are represented by coordinate vectors with respect to a normal basis, we call this a *normal basis representation* for the field.

In a normal basis representation, the operation of squaring a field element is extremely simple. For example, suppose $\{\beta, \beta^2, \beta^4, \cdots, \beta^{2^{n-1}}\}$ is a normal basis, and denote $\beta_i = \beta^{2^i}$, for $0 \leqq i \leqq n-1$. A field element a can be expressed as $a = \sum_{i=0}^{n-1} a_i \beta_i$, $a_i = 0$ or 1, for $0 \leqq i \leqq n-1$. Then, $a^2 = \sum_{i=0}^{n-1} a_{i-1} \beta_i$, where the subscripts are reduced modulo n. That is, the coordinate vector for a^2 is computed by a cyclic shift of the coordinate vector for a.

The Massey-Omura algorithm [11] for multiplication in GF(2^n) uses a normal basis representation. This algorithm has been studied further by several researchers. In particular, we should mention the use of the so-called *optimal normal bases* of Mullin et al. [12]. An optimal normal basis is a special type of normal basis that allows for a very efficient hardware implementation of the Massey-Omura algorithm. For more information, we refer the reader to [12].

The algorithms for exponentiation that we study will also make use of a normal basis representation. All necessary multiplications would be done using the Massey-Omura algorithm. For the purposes of this paper, the details of the Massey-Omura

* Received by the editors March 6, 1989; accepted for publication (in revised form) November 15, 1989. This research was supported by Natural Sciences and Engineering Research Council of Canada operating grant A9287.

† Department of Computer Science, University of Manitoba, Winnipeg, Manitoba, Canada R3T2N2. Present address, Computer Science and Engineering, University of Nebraska, Lincoln, Nebraska 68588-0115.

algorithm are unimportant. However, since squaring in a normal basis representation is accomplished by a cyclic shift, we shall assume throughout this paper that the time required to square any field element is negligible.

Suppose that we want to compute $a^e \in \mathrm{GF}(2^n)$, where $e = \sum_{i=0}^{n-1} e_i 2^i$, $e_i = 0$ or 1, for $0 \leqq i \leqq n-1$. Then, $a^e = \prod_{i=0}^{n-1} a^{e_i 2^i}$. The *Hamming weight* of e is $\mathrm{wt}(e) = \sum_{i=0}^{n-1} e_i$, i.e., the number of nonzero coordinates in the vector e. Then a^e can be calculated using $\mathrm{wt}(e)-1$ multiplications (ignoring the time required to perform the necessary squarings). Since $\mathrm{wt}(e) \leqq n$, at most $n-1$ multiplications are required. However, on average, $\mathrm{wt}(e) = n/2$, so the calculation of a^e requires (on average) $n/2-1$ multiplications. This method of exponentiation will be referred to as the *binary method.*

In the remainer of the paper we investigate algorithms for exponentiation that require fewer multiplications than the binary method. We also discuss the parallelization of these algorithms.

2. A divide-and-conquer algorithm. In this section, we describe the technique of Agnew et al. [1], [2], which is essentially a divide-and-conquer algorithm. Let k be an integer, and let $s = \lceil n/k \rceil$. Recall that we want to evaluate a^e. Write the exponent e as $e = \sum_{i=0}^{s-1} w_i 2^{ki}$, where $0 \leqq w_i \leqq 2^k - 1$, for $0 \leqq i \leqq s-1$. Then, e can be rewritten as

$$e = \sum_{w=1}^{2^k-1} \left(\sum_{\{i: w_i = w\}} 2^{ki} \right) \cdot w = \sum_{w=1}^{2^k-1} \lambda(w) \cdot w,$$

where $\lambda(w) = \sum_{\{i: w_i = w\}} 2^{ki}$.

Example 1. Suppose $n = 12$, $k = 2$, and $e = 1499 = 2^{10}+2^8+2^7+2^6+2^4+2^3+2^1+2^0$. Then $s = 6$, and e would be expressed as

$$e = (2^{10}+2^8+2^4)1 + (2^2)2 + (2^6+2^0)3,$$

so

$$\lambda(1) = 2^{10}+2^8+2^4, \quad \lambda(2) = 2^2, \quad \lambda(3) = 2^6+2^0.$$

Then, as in [1], we compute a^e to be

$$a^e = \prod_{w=1}^{2^k-1} a^{\lambda(w) \cdot w} = \prod_{w=1}^{2^k-1} (a^w)^{\lambda(w)}.$$

We shall compute a^e in two phases, as indicated in the following algorithm.

ALGORITHM **d&c-exponentiate (a, e, k).**
{compute a^e}
Phase 1: compute the $2^k - 1$ terms a^w $(1 \leqq w \leqq 2^k - 1)$.
Phase 2: multiply the terms $(a^w)^{\lambda(w)}$ together.

In Example 1, a^{1499} would be computed as

$$a^{1499} = (a^1)^{2^{10}} \cdot (a^1)^{2^8} \cdot (a^1)^{2^4} \cdot (a^2)^{2^2} \cdot (a^3)^{2^6} \cdot (a^3)^{2^0}.$$

Here, we would require one multiplication in Phase 1, to compute a^3, and five multiplications in Phase 2, to multiply the six terms together. This compares to $\mathrm{wt}(1499) - 1 = 7$ multiplications required by the binary method.

Let us now investigate how many multiplications are required in general during each phase of the computation.

In Phase 2, the total number of terms to be multiplied together is at most $s = \lceil n/k \rceil$. The product of these s terms will be computed using $s-1$ multiplications.

In Phase 1, we must compute all elements a^w $(1 \leqq w \leqq 2^k - 1)$. Let us define $M(k)$ to be the total number of multiplications required to compute all these elements. It is immediate that $M(k) \leqq 2^k - 1$, for we can compute each a^w from a^{w-1} by multiplying by a. In fact, $M(k) \leqq 2^{k-1} - 1$, by means of the following algorithm:

ALGORITHM **compute-small-powers (a, k).**
{compute a^w for $1 \leqq w \leqq 2^k - 1$}
$a^1 := a$;
for $w := 2$ to $2^k - 1$ do
    if w is even then {perform a cyclic shift of $a^{w/2}$}
        compute a^w as $(a^{w/2})^2$
    if w is odd then
        compute a^w as $a^{w-1} \times a$;

Let us now return to our algorithm **d&c-exponentiate** for computing a^e. Using the above bound for $M(k)$, we have the following bound.

THEOREM 2.1. *Suppose elements of* GF(2^n) *are represented by coordinate vectors with respect to a normal basis. Suppose* $a \in \mathrm{GF}(2^n)$ *and* $0 \leqq e \leqq 2^n - 1$. *Let* $1 \leqq k \leqq n$ *be any integer. Then algorithm* **d&c-exponentiate** *can compute* a^e *in at most* $2^{k-1} + \lceil n/k \rceil - 2$ *multiplications.*

We are free to choose k as we wish, so we would choose it so as to minimize the number of multiplications. If we take k to be roughly $\log_2 n - \log_2 \log_2 n$, then the number of multiplications is approximately

$$\frac{n}{2 \log_2 n} + \frac{n}{\log_2 n - \log_2 \log_2 n} = O\left(\frac{n}{\log_2 n - \log_2 \log_2 n}\right),$$

which is certainly an asymptotic improvement over $n/2$.

3. Parallel algorithms. Suppose n is even, and we have $n/2$ parallel processors. Then, we can simultaneously multiply n elements in pairs, producing $n/2$ elements. These can then be multiplied together in pairs, in a similar fashion. Continuing this process, we obtain the value of a^e after $\lceil \log_2 n \rceil$ rounds. Similarly, if n is odd, then we can multiply n elements in $\lceil \log_2 n \rceil$ rounds using $\lfloor n/2 \rfloor$ processors. This is considerably faster than a sequential algorithm, but we do require a considerable number of processors. This technique will be called *binary fan-in multiplication.*

We are interested in the following two questions in this section. First, can we perform an exponentiation in $\lceil \log_2 n \rceil$ rounds using fewer than $n/2$ parallel processors? Second, can we obtain a more modest, but still significant speedup using fewer processors? We answer both questions in the affirmative using a parallel implementation of algorithm **d&c-exponentiate.**

First, recall the two phases of algorithm **d&c-exponentiate.**

In Phase 2, the total number of terms to be multiplied together is at most $s = \lceil n/k \rceil$. The product of these s terms will be computed using binary fan-in multiplication. If we have $\lfloor s/2 \rfloor$ processors, then $\lceil \log_2 s \rceil$ rounds will be required.

In Phase 1, we must compute all elements a^w $(1 \leqq w \leqq 2^k - 1)$. It does not seem very promising to parallelize algorithm **compute-small-powers,** so we shall consider other algorithms. Clearly, we can compute any particular element a^w $(1 \leqq w \leqq 2^k - 1)$ in at most k rounds using binary fan-in multiplication. Therefore, with sufficient processors, we can compute all elements a^w $(1 \leqq w \leqq 2^k - 1)$ in k rounds. If we compute every element a^w $(1 \leqq w \leqq 2^k - 1)$ independently using binary fan-in multiplication, a fairly simple calculation shows that $(k-1)2^{k-2}$ parallel processors are needed. We can

do better by using another divide-and-conquer algorithm. Consider the following algorithm, which is similar to the algorithm POWERS in [14].

ALGORITHM **d&c-compute-small-powers (a, k).**
{compute a^w for $1 \leqq w \leqq 2^k - 1$ in parallel}
Phase 1: compute the $2^j - 1$ terms a^w $(1 \leqq w \leqq 2^j - 1)$, where $j = \lceil k/2 \rceil$.
Phase 2: for each term a^u $(1 \leqq u \leqq 2^{k-j} - 1)$ computed in Phase 1, compute $(a^u)^{2^j}$.
Phase 3: multiply every term a^w computed in Phase 1 by every term $(a^u)^{2^j}$ computed in Phase 2.

As an example, suppose we call **d&c-compute-small-powers (a, 5).** Then, $j = 3$. In Phase 1, we would compute a^1, a^2, a^3, a^4, a^5, a^6, and a^7. In Phase 2, we would compute a^8, a^{16}, and a^{24}. Finally, in Phase 3, we would compute 21 products of elements from Phase 1 with elements from Phase 2.

In general, the elements computed in Phases 1, 2, and 3 comprise all a^w $(1 \leqq w \leqq 2^k - 1)$. If we have enough processors to perform all calculations in parallel, how many rounds of multiplication are required? Denote the number of rounds required by algorithm **d&c-compute-small-powers (a, k)** by $R(k)$, and the number of parallel processors required by $P(k)$. The number of rounds required in Phase 1 is $R(j)$, if we call the same algorithm recursively. No multiplications are required in Phase 2 (only cyclic shifts are done). Finally, Phase 3 requires only one round. Hence, $R(k)$ satisfies the recurrence

$$R(2k) = R(k) + 1 \quad \text{if } k > 0,$$
$$R(2k+1) = R(k+1) + 1 \quad \text{if } k > 0,$$
$$R(1) = 0.$$

It is easy to prove by induction that the solution to this recurrence is $R(k) = \lceil \log_2 k \rceil$. It is also easy to see that the number of processors required is

$$P(k) = (2^j - 1)(2^{k-j} - 1) = (2^{k/2} - 1)^2 \quad \text{if } k \text{ is even}$$
$$= (2^{(k+1)/2} - 1)(2^{(k-1)/2} - 1) \quad \text{if } k \text{ is odd.}$$

Hence, $P(k) < 2^k$, a significant improvement over $(k-1) \cdot 2^{k-2}$ for large k.
Now, if we use algorithm **d&c-compute-small-powers** in Phase 1 of algorithm **d&c-exponentiate,** the total number of rounds is $\lceil \log_2 k \rceil + \lceil \log_2 s \rceil$, where $s = \lceil n/k \rceil$, provided the number of parallel processors is at least $\max\{P(k), \lfloor s/2 \rfloor\}$. So we have proved the following.

THEOREM 3.1. *Suppose elements of* GF(2^n) *are represented by coordinate vectors with respect to a normal basis. Suppose* $a \in \mathrm{GF}(2^n)$ *and* $0 \leqq e \leqq 2^n - 1$. *Let* $1 \leqq k \leqq n$ *be an integer, and define* $s = \lceil n/k \rceil$. *If we have at least* $\max\{P(k), \lfloor s/2 \rfloor\}$ *parallel processors, then algorithm* **d&c-exponentiate** *can compute* a^e *in at most* $\lceil \log_2 k \rceil + \lceil \log_2 s \rceil$ *rounds of multiplications.*

It is easy to see that $\lceil \log_2 n \rceil \leqq \lceil \log_2 k \rceil + \lceil \log_2 s \rceil \leqq \lceil \log_2 n \rceil + 1$; hence the number of rounds is at most one more than required by binary fan-in multiplication. However, if we choose k to be approximately $\log_2 n - \log_2 \log_2 n$, then the number of processors required is approximately $n/(\log_2 n - \log_2 \log_2 n)$, which is asymptotic improvement over $n/2$.

If $P(k) \geqq \lfloor s/2 \rfloor$, then we require $P(k)$ processors in Theorem 3.1. Let us now examine what we can do with $P(k)$ processors when $P(k) \leqq \lfloor s/2 \rfloor$. Phase 1 of algorithm **d&c-exponentiate** operates as before, but now we do not have enough processors to

perform the binary fan-in multiplication in Phase 2. However, we can modify binary fan-in multiplication to work with fewer processors (but more rounds will be required).

Define $F(m, p)$ to be the number of rounds required to multiply m elements using p parallel processors. We have already observed that $F(m, \lfloor m/2 \rfloor) = \lceil \log_2 m \rceil$. Using a straightforward modification of binary fan-in multiplication, it is not difficult to see that

$$F(m, p) = \left\lceil \frac{m - 2p_0}{p} \right\rceil + \log_2 p_0 + 1,$$

where p_0 = the largest power of two not exceeding p.

This is obtained as follows. There are a total of $m-1$ multiplications to be performed. The last $\log_2 2p_0 = \log_2 p_0 + 1$ rounds comprise a binary fan-in multiplication of $2p_0$ elements, The total number of multiplications performed in these $\log_2 p_0 + 1$ rounds is $2p_0 - 1$. Hence, in the earlier rounds, there are $m - 2p_0$ multiplications to be performed. It is not difficult to see that we can perform $m - 2p_0$ multiplications in $\lceil (m-2p_0)/p \rceil$ rounds by the p processors, in such a way that we are left with $2p_0$ elements to be multiplied together in the last $\log_2 p_0 + 1$ rounds.

For example, suppose we want to compute $\prod_{i=1}^{9} x_i$ using three processors. Then $p_0 = 2$, $F(9, 3) = 4$, and the product can be computed as follows:

Round 1 compute x_1x_2 and x_8x_9.
Round 2 compute $(x_1x_2)x_3$, x_4x_5, and x_6x_7.
Round 3 compute $(x_1x_2x_3)(x_4x_5)$ and $(x_6x_7)(x_8x_9)$.
Round 4 compute $(x_1x_2x_3x_4x_5)(x_6x_7x_8x_9)$.

We now apply this technique in Phase 2 of algorithm **d&c-exponentiate**. For the sake of simplicity, we shall assume that we have 2^k processors (recall that $P(k) < 2^k$). We obtain the following result.

THEOREM 3.2. *Suppose elements of* GF(2^n) *are represented by coordinate vectors with respect to a normal basis. Suppose* $a \in \mathrm{GF}(2^n)$ *and* $0 \leqq e \leqq 2^n - 1$. *Let* $1 \leqq k \leqq n$ *be an integer, and define* $s = \lceil n/k \rceil$. *Suppose we have* 2^k *parallel processors, where* $2^k \leqq \lfloor s/2 \rfloor$. *Then algorithm* **d&c-exponentiate** *can compute* a^e *in at most* $\lceil \log_2 k \rceil + \lceil s/2^k \rceil + k - 1$ *rounds of multiplications.*

Proof. The number of rounds required is $\lceil \log_2 k \rceil + F(s, 2^k) = \lceil \log_2 k \rceil + \lceil s/2^k \rceil + k - 1$. □

This also compares favourably with the algorithm given in [1], where the number of rounds of multiplications is (on average) $2^k + \lceil s/2^k \rceil + k/2 - 4$.

4. An example: GF(2^{593}). To illustrate our algorithms, let us look at GF(2^{593}). This is an interesting example to study, since hardware to perform arithmetic in GF(2^{593}) has been developed (see [4]).

First, we consider the sequential algorithm of § 2. The number of multiplications required is $2^{k-1} + \lceil n/k \rceil - 2$ (Theorem 2.1). For various values of k, we obtain the information in Table 1. Thus the minimum is attained when $k = 6$. A maximum of 129 multiplications suffice to exponentiate in GF(2^{593}), as compared to the average of 291.5 and the maximum of 592 in the binary method.

When we incorporate parallelism, the number of rounds can be reduced to as few as 10. We determine the number of processors required by Theorem 3.1 (see Table 2). Hence, we can exponentiate in 10 rounds with 49 processors, a considerable reduction from the 297 required for binary fan-in multiplication.

Finally, we determine the number of rounds required by fewer processors, using the bound of Theorem 3.2. The information is summarized in Table 3. So, we can

TABLE 1

k	$2^{k-1}+\lceil n/k \rceil -2$
2	297
3	200
4	155
5	133
6	129
7	147

TABLE 2

k	s	No. of rounds	$P(k)$	$\lfloor s/2 \rfloor$	No. of processors
2	297	10	1	143	143
3	198	11	3	99	99
4	149	10	9	75	75
5	119	10	21	60	60
6	99	10	49	49	49
7	85	10	105	42	105

TABLE 3

k	s	2^k	$\lceil \log_2 k \rceil + \lceil s/2^k \rceil + k - 1$
2	297	4	77
3	198	8	29
4	149	16	15
5	119	32	11

exponentiate in 77 rounds with 4 processors; in 29 rounds with 8 processors; in 15 rounds with 16 processors; or in 11 rounds with 32 processors.

5. Comments and summary. We have described sequential and parallel algorithms for exponentiating in GF (2^n) using a normal basis representation. These algorithms are adaptive, in that we can vary the number of processors to suit our needs. They should be very practical for realistic applications of arithmetic in GF (2^n) for values of n up to 1,000 and beyond.

We can also use the algorithms in this paper to compute multiplicative inverses in GF (2^n), using the formula $a^{-1} = a^{2^n-2}$. Other algorithms for finding inverses in GF (2^n) using a normal basis representation are presented in [2] and [8].

We should also mention that several other approaches to parallel exponentiation in finite fields and related problems have been investigated in recent years. See, for example, [3], [6], [7], [10], [15], and [16]. Most of these papers discuss the existence of Boolean circuits for the given algorithms. In [6], for example, Eberly shows that we can construct a Boolean circuit of depth $O(\log n)$ and size $n^{O(1)}$ to exponentiate in GF (2^n). This is a very interesting and important result from a theoretical standpoint. However, no precise upper bounds are given on the size of the resulting circuits, and it does not seem practical to construct such circuits for large values of n.

For purposes of comparison, we can give fairly precise upper bounds on the size of circuits to implement the algorithms in this paper. The Massey-Omura multiplication algorithm can be implemented by a Boolean circuit of depth $O(\log n)$ and size $O(n^3)$. Moreover, if an optimal normal basis exists in GF(2^n) (see [12]), then the size can be reduced to $O(n^2)$. Using these Boolean circuits, it is not difficult to see that the parallel version of our exponentiation algorithm can be realized (in general) by a circuit of depth $O(\log^2 n)$ and size $O(n^4/(\log n - \log\log n))$. If an optimal normal basis exists, then the size is reduced to $O(n^3/(\log n - \log\log n))$.

Finally, we remark that the techniques of this paper do not appear to be readily applicable to exponentiation in GF(p^n), when $p > 2$. The reason for this is as follows. A normal basis in GF(p^n) is one having the form $\{\beta, \beta^p, \beta^{p^2}, \cdots, \beta^{p^{n-1}}\}$. Hence, using a normal basis representation, the pth power of an element is given by a cyclic shift. This does not seem to be of much help in exponentiating if $p > 2$.

REFERENCES

[1] G. B. AGNEW, R. C. MULLIN, AND S. A. VANSTONE, *Fast exponentiation in* GF(2^n), in Advances in Cryptology—Eurocrypt '88, Lecture Notes in Computer Science Vol. 330, Springer-Verlag, Berlin, 1988, pp. 251-255.

[2] G. B. AGNEW, T. BETH, R. C. MULLIN, AND S. A. VANSTONE, *Arithmetic operations in* GF(2^n), J. Cryptology, submitted.

[3] P. BEAME, S. COOK, AND H. J. HOOVER, *Log depth circuits for division and related problems*, SIAM J. Comput., 13 (1984), pp. 268-276.

[4] CA34C168 *data encryption processor*, Document 01-34168-500, Calmos Semiconductor Inc, Kanata, Ontario, Canada, June 1988.

[5] W. DIFFIE AND M. HELLMAN, *New directions in cryptography*, IEEE Trans. Inform. Theory, 22 (1976), pp. 644-654.

[6] W. EBERLY, *Very fast parallel polynomial arithmetic*, SIAM J. Comput., 18 (1989), pp. 955-976.

[7] F. E. FICH AND M. TOMPA, *The parallel complexity of exponentiating polynomials over finite fields*, J. Assoc. Comput. Mach., 35 (1988), pp. 651-667.

[8] T. ITOH, O. TEECHAI, AND S. TSUJII, *A fast algorithm for computing multiplicative inverses in* GF(2^t) *using normal bases*, J. Soc. Electronic Commun. (Japan), 44 (1986), pp. 31-36. (In Japanese.)

[9] R. LIDL AND H. NIEDERREITER, *Finite Fields*, Cambridge University Press, London, 1987.

[10] B. E. LITOW AND G. I. DAVIDA, *$O(\log(n))$ parallel time finite field inversion*, In Proc. 3rd Aegean Workshop on Computing, Corfu, Lecture Notes in Computer Science Vol. 319, Springer-Verlag, Berlin, 1988, pp. 74-80.

[11] J. L. MASSEY AND J. K. OMURA, *Computational method and apparatus for finite field arithmetic*, U.S. Patent Application, 1981.

[12] 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.

[13] A. M. ODLYZKO, *Discrete logarithms and their cryptographic significance*, in Advances in Cryptology—Eurocrypt '84, Lecture Notes in Computer Science Vol. 209, Springer-Verlag, Berlin, 1985, pp. 224-314.

[14] F. P. PREPARATA AND D. V. SARWATE, *An improved parallel processor bound in fast matrix inversion*, Inform. Process. Lett., 7 (1978), pp. 148-150.

[15] J. H. REIF, *Logarithmic depth circuits for algebraic functions*, SIAM J. Comput., 15 (1986), pp. 231-242.

[16] J. VON ZUR GATHEN, *Inversion in finite fields using logarithmic depth*, J. Symbolic Comput., to appear.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 718-727, August 1990

011

TOWARD UNDERSTANDING EXCLUSIVE READ*

FAITH E. FICH† AND AVI WIGDERSON‡

Abstract. The ability of many processors to simultaneously read from the same cell of shared memory can give additional power to a parallel random access machine. In this paper, a natural Boolean function of n variables is described, and it is shown that the expected running time of any probabilistic EROW PRAM computing this function is in $\Omega(\sqrt{\log n})$, although it can be computed by a CROW PRAM in $O(\log \log n)$ steps.

Key words. shared memory parallel computation, PRAM, lower bounds, exclusive read, decision trees

AMS(MOS) subject classifications. 68Q05, 68Q10

1. Introduction.

In [8], Snir proved that the following range search problem has time complexity $\Theta(\sqrt{\log n})$ on an EREW PRAM:

> Given distinct inputs $x_1, \cdots, x_n$, and y, with $x_1 < \cdots < x_n$, determine the maximum index i such that $x_i < y$.

This problem can be solved in a constant number of steps on a CREW PRAM. Thus, in certain situations, CREW PRAMs are more powerful than EREW PRAMs.

But this result does not tell us everything we would like to know. For example, consider the relationship between the CROW and CREW PRAMs. The OR of n Boolean values, at most one of which is 1, can be determined in a constant number of steps on a CREW PRAM, but $\log_2 n$ steps are required on a CROW PRAM [2]. In contrast, Nisan [7] proved that any Boolean function $f: \{0, 1\}^n \to \{0, 1\}$ has, to within a small constant factor, the same time complexity on CREW and CROW PRAMs.

Two features of Snir's result are important in this regard. The first is that, like the restricted version of OR in the previous paragraph, the domain of his range search problem is not complete. (A complete domain is one of the form D^n for some set D.) Such a situation can be viewed as having information about the inputs built into the program. This information can be used by the algorithm to ensure that no conflict arises during a potentially concurrent read or write. In particular, it enables the range search problem to be solved quickly on a CREW PRAM. Gafni, Naor, and Ragde [5] recently improved Snir's result by exhibiting a function with a complete domain that is easy to solve on a CROW PRAM and still difficult to solve on an EREW PRAM.

Another feature of these results is that the proofs of the lower bounds depend on the domains of the functions being very large. The essential idea is to show that, using Ramsey theory, there is a large subset of the domain for which the states of all processors and the contents of all shared memory cells at each point in the computation depend only on the relative order of the input values, not on their values.

* Received by the editors September 17, 1989; accepted for publication (in revised form) November 30, 1989. This work was partially supported by the Information Technology Research Centre of Ontario, Ontario, Canada.

† University of Toronto, Department of Computer Science, Toronto, Ontario, Canada M5S 1A4. The research of this author was partially supported by Natural Sciences and Engineering Research Council of Canada grant A9176 and a grant from the University of Toronto, Toronto, Ontario, Canada.

‡ Hebrew University of Jerusalem, Computer Science Department, Jerusalem 91904, Israel. The research of this author was partially supported by an Alon Fellowship and the American-Israeli Binational Science Foundation.

It remains open whether all Boolean functions can be computed as quickly by an EREW PRAM as by a CREW PRAM. We make progress toward solving this problem by defining a natural Boolean function that can be computed quickly on a CROW PRAM and we prove that it requires a long time to solve on an EROW PRAM, even if the algorithm is allowed to make probabilistic choices. A new probabilistic technique is used to obtain this lower bound. We also give evidence that this function is hard to solve on an EREW PRAM. Finally, we explain where the difficulties arise when attempting to extend the lower bound to the EREW PRAM.

2. Models. In this paper, we consider nonuniform parallel random access machines (PRAMs) with an infinite number of processors and shared memory cells that can contain arbitrarily large values. The n input values initially appear in the first n cells of shared memory and the answer is the contents of the first shared memory cell at the end of the computation. The processors work together synchronously to solve a problem. At each step, a processor may read from one cell of shared memory, then perform an arbitrary amount of local computation, and finally write to one cell of shared memory.

In the concurrent read, exclusive write (CREW) PRAM, multiple processors may not write to the same memory cell at the same step of a computation, although any number of processors may simultaneously read from a single cell. The exclusive read, exclusive write (EREW) PRAM does not allow simultaneous access to a shared memory cell for either reading or writing.

Complete networks of processors are also interesting models of parallel computation. Again we assume that there is an infinite number of processors and they work synchronously. At each step, a processor reads the message posted by one processor of its choice, performs an arbitrary amount of local computation, and then posts a new message. The input to a problem is initially distributed among the first n processors and, at the end of the computation, the first processor has determined the answer.

This model is equivalent to a restricted version of the CREW PRAM in which there is a one-to-one correspondence between processors and shared memory cells and only the processor corresponding to a particular memory cell may write to it. Many algorithms designed for CREW PRAMs avoid write conflicts in this way. Dymond and Ruzzo, who introduced this model in [3], call it the concurrent read, owner write (CROW) PRAM.

If we further restrict the CROW PRAM so that, at each step, at most one processor can read from each shared memory cell, we obtain the exclusive read, owner write (EROW) PRAM. This is the model for which we prove a lower bound. The EROW PRAM corresponds to a complete network in which each posted message can be read by at most one processor at a time. (Note that a processor is not required to know which processor, if any, is reading its message at a given step.)

In many respects, these models are more powerful than any realistic parallel machine. However, this does not affect the significance of the lower bounds we obtain. On the other hand, the algorithms presented in this paper are quite simple and easily implemented on less powerful models.

For our lower bound proof, it is necessary to introduce the concept of a CROW PRAM processor *knowing* certain input bits. The processors' knowledge is defined inductively for each step of the computation. Initially, each of the first n processors knows the input bit it has been given (i.e., the ith processor knows the ith input bit). No processor knows any other bit of the input. The input bits known by a processor after it reads the message posted by another processor are the union of the bits known

by the two processors before the read occurred. Changing the value of any input bits that a processor does not know at any given point in time cannot change the state of the processor at that time.

The following two lemmas describe properties of knowledge that are important for our lower bound proof.

LEMMA 1 (Cook, Dwork, and Reischuk [2]). *For a* CROW PRAM, *on any input, every processor knows at most* 2^t *input bits immediately after step t.*

LEMMA 2 (Beame [1]). *For an* EROW PRAM, *on any input, each input bit is known by at most* 2^t *processors immediately after step t.*

3. The Boolean decision-tree evaluation problem. Suppose we are given a decision tree, each node of which is labeled by a Boolean variable (called a *query*). Suppose we are also given an outcome for each query. Consider the path that starts at the root, goes left whenever the query at the current node has outcome 0 and goes right whenever the query has outcome 1. The decision-tree evaluation problem is to determine the outcome of the query labeling the leaf reached by this path.

For example, given the decision tree in Fig. 1 and the outcomes $x_0 = 1$, $x_1 = 1$, $x_2 = 1$, and $x_3 = 0$, the second leaf from the left is reached and, hence, the decision tree has value 1.

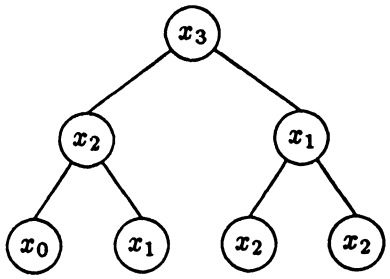


FIG. 1. *A decision tree.*

Let $D_{m,h}: \{0,1\}^{m(2^{h+1}-1)+2^m} \to \{0,1\}$ be a Boolean function representing a Boolean decision-tree evaluation problem for a complete binary tree of height h in which every node is labeled by one of 2^m queries. This can be done by dividing the first $m(2^{h+1}-1)$ input bits into $2^{h+1}-1$ blocks of length m. The value y_i of the ith block denotes the index of the query labeling the ith node in the tree. The last 2^m bits, $x_0, \cdots, x_{2^m-1}$, denote the outcomes of the queries. For example, $D_{2,2}(111001000110101110) = 1$. (See Fig. 1.)

Clearly, the Boolean function $D_{m,h}$ can be computed by a (sequential) random access machine in $(m+1)h$ steps, following the path through the tree, alternately reading the label of the next node and then the outcome of the query labeling it.

Using an EROW PRAM, an $O(\log m + h)$ upper bound can be obtained by first collecting the m bits comprising the label of each node into one memory cell, using $O(m/\log m)$ processors and $O(\log m)$ time. This is done in parallel for each node in the tree. Then the path through the tree can be followed in $O(h)$ more steps.

This can be improved to $O(\log m + \log h)$ on a CROW PRAM. After the bits of the node labels have been collected, as above, one processor is assigned to each node of the decision tree. In one step, each processor reads the outcome of the query labeling its node. Because many nodes in the decision tree can have the same label, concurrent

read is essential for this step. The outcome of the query labeling a node defines a pointer from the node to one of its two children. Using pointer jumper [6], the unique path from the root to a leaf can then be determined in $O(\log h)$ steps, even by an EROW PRAM.

Another (although less efficient) CROW PRAM algorithm creates a table of values for the function computed by the decision tree and then performs table look up using the query outcomes. For each of the 2^{2^m} possible outcomes for the queries, a group of $2^{h+1}-1$ processors is allocated. In one step, each group of processors makes a copy of the decision tree. Using pointer jumping, as in the previous algorithm, the processors in each group determine the answer that would be obtained assuming the query outcomes associated with their group. (The actual query outcomes have not been read at this point in the algorithm.) In $O(m)$ steps, the correct group can be determined from the actual outcomes of the 2^m queries. Then the answer can be determined by reading from the appropriate place in the table. The total time taken by this algorithm is $O(m+\log h)$.

Only the first step of this algorithm uses concurrent read. With exclusive read, the 2^{2^m} copies of the decision tree can be constructed in 2^m steps. This gives rise to an $O(2^m+\log h)$ upper bound on the EROW PRAM.

4. The lower bound.

THEOREM 3. *The expected number of steps performed by a probabilistic* EROW PRAM *to solve the Boolean decision tree evaluation problem for* $m=3T$ *and* $h=6T^2$ *is more than* $T/2$.

Proof. Let $X_0, \cdots, X_{2^m-1}$ be random variables whose values are independently and uniformly chosen from the range $\{0,1\}$. The sequence of random variables $X=(X_0, \cdots, X_{2^m-1})$ is used to denote the outcomes of the queries. Let $Y_1, \cdots, Y_{2^{h+1}-1}$ be random variables with range $\{0, \cdots, 2^m-1\}$. The sequence $Y=(Y_1, \cdots, Y_{2^{h+1}-1})$ is used to denote a labeling of the nodes in the decision tree. The label of the root (i.e., the value of the random variable corresponding to the root) is chosen using a uniform distribution. Once all ancestors of a node have been labeled, the label of the node is chosen uniformly among those queries not labeling any of its ancestors. This gives us a uniformly chosen labeling of the decision tree with the property that all nodes along any path from the root to a leaf are labeled by different queries. It suffices to show [9] that the average number of steps (with respect to this input distribution) performed by any deterministic EROW PRAM solving this problem is more than $T/2$.

Imagine the decision tree sliced horizontally into T pieces, each of height $k=6T$. For $t=0, \cdots, T$, the node at depth kt on the path determined by the query outcomes X in the decision tree labeled by Y is denoted by $R_t(Y,X)$ and the subtree rooted at $R_t(Y,X)$ is denoted by $S_t(Y,X)$. In particular, $S_0(Y,X)$ is the entire decision tree, $R_0(Y,X)$ is its root, and $R_T(Y,X)$ is the leaf that is reached. This is illustrated in Fig. 2. Finally, let $U_t(Y,X)$ denote the set of queries that do not label any proper ancestor of $R_t(Y,X)$. Then $U_0(Y,X)$ is the set of all 2^m queries, $U_0(Y,X) \supseteq U_1(Y,X) \supseteq \cdots \supseteq U_T(Y,X)$, and $|U_t(Y,X)| = 2^m - kt$.

CLAIM. *The probability that there is a processor that, at the end of step* t, *knows both the outcome of a query in* $U_t(Y,X)$ *and* (*any bit of*) *the label of a node in the subtree* $S_t(Y,X)$ *is at most* $t2^{12-T}$.

In particular, from the claim, the probability is at most $T2^{12-T}$ that processor P_1 (which is supposed to determine the answer) knows both the outcome of a query in $U_T(Y,X)$ and (a bit of) the label of $R_T(Y,X)$ within T steps. We will show that the probability that processor P_1 has the correct answer is only slightly more than $\frac{1}{2}$.

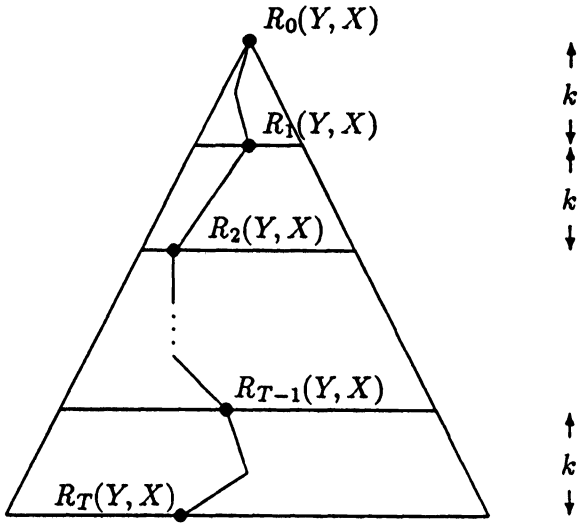


FIG. 2. *A decision tree sliced into pieces.*

Let C denote the event that processor P_1 has determined the correct answer within T steps, let B denote the event that P_1 knows the label of $R_T(Y, X)$ within T steps, and let A denote the event that the label of $R_T(Y, X)$ is a query whose outcome is known by P_1 within T steps. If the label of $R_T(Y, X)$ is a query whose outcome is not known by P_1, then changing only the outcome of this query does not change P_1's state; although, for the algorithm to be correct, it should. Since the outcome of the query labeling $R_T(Y, X)$ is equally likely to be 0 or 1, it follows that

$$\text{pr}[C|\bar{A}] \leqq \tfrac{1}{2}.$$

If P_1 does not know the label of $R_T(Y, X)$, changing the label of $R_T(Y, X)$ to anything else in $U_T(Y, X)$ does not change P_1's state. Because the label of $R_T(Y, X)$ is equally likely to be any query in $U_T(Y, X)$ and, by Lemma 1, P_1 knows the outcomes of at most 2^T queries,

$$\text{pr}[A|\bar{B}] \leqq \frac{2^T}{|U_T(Y, X)|} = \frac{2^T}{2^{3T} - 6T^2} \leqq 2^{2-2T}.$$

Thus

$$\begin{aligned}\text{pr}[C] &= \text{pr}[C|A \wedge B] \cdot \text{pr}[A \wedge B] + \text{pr}[C|\bar{A}] \cdot \text{pr}[\bar{A}] + \text{pr}[C|A \wedge \bar{B}] \cdot \text{pr}[A \wedge \bar{B}] \\ &\leqq 1 \cdot \text{pr}[A \wedge B] + \text{pr}[C|\bar{A}] \cdot 1 + 1 \cdot \text{pr}[A|\bar{B}] \\ &\leqq \tfrac{1}{2} + T2^{12-T} + 2^{2-2T}.\end{aligned}$$

A correct algorithm always performs at least one step. If the correct answer has not been determined within T steps, the algorithm must perform at least one more step. Therefore, the expected number of steps performed by a correct algorithm is at least

$$\begin{aligned}\text{pr}[C] \cdot 1 + \text{pr}[\bar{C}] \cdot (T+1) &= 1 + T(1 - \text{pr}[C]) \\ &\geqq 1 + T/2 - T^2 2^{12-T} - T2^{2-2T} \\ &> T/2 \quad \text{for } T \geqq 21.\end{aligned}$$

Proof of the Claim. Let $Q_t(Y, X)$ denote the set of processors that know the outcome of a query in $U_t(Y, X)$ immediately after step t. (Since there are 2^m queries and, by Lemma 2, the outcome of no query is known by more than 2^t processors immediately after step t, it follows that $|Q_t(Y, X)| \leqq 2^{m+t}$.) Similarly, let $L_t(Y, X)$ denote the set of processors that know the label of some node in $S_t(Y, X)$ immediately after step t. We prove by induction on t that

$$\text{pr}[Q_t(Y, X) \cap L_t(Y, X) \neq \phi] \leqq t2^{12-T}.$$

Before the first step, each processor knows at most one input bit. Hence the claim is true for $t = 0$. Now assume the claim is true for t, where $0 \leqq t < T$. Then

$$\begin{aligned}&\text{pr}[Q_{t+1}(Y, X) \cap L_{t+1}(Y, X) \neq \phi] \\ &\quad \leqq \text{pr}[Q_t(Y, X) \cap L_t(Y, X) \neq \phi] \\ &\qquad + \text{pr}[Q_{t+1}(Y, X) \cap L_{t+1}(Y, X) \neq \phi \mid Q_t(Y, X) \cap L_t(Y, X) = \phi].\end{aligned}$$

By the induction hypothesis,

$$\text{pr}[Q_t(Y, X) \cap L_t(Y, X) \neq \phi] \leqq t2^{12-T}.$$

Therefore, we suppose $Q_t(Y, X) \cap L_t(Y, X) = \phi$.

If $Q_{t+1}(Y, X) \cap L_{t+1}(Y, X) \neq \phi$ then either

(1) There is a processor (in $L_t(Y, X)$) that knows the label of a node in $S_{t+1}(Y, X)$ at the end of step t and, at step $t+1$, reads from a processor in $Q_t(Y, X)$, or

(2) There is a processor in $Q_t(Y, X)$ that, at step $t+1$, reads from a processor (in $L_t(Y, X)$) that knows the label of a node in $S_{t+1}(Y, X)$ at the end of step t.

We handle these two cases one at a time.

First, consider the set of processors in $L_t(Y, X)$ that, at step $t+1$, read from processors in $Q_t(Y, X)$. Each processor in $Q_t(Y, X)$ can have its message read by at most one processor at step $t+1$, so there are at most $|Q_t(Y, X)| \leqq 2^{m+t}$ such processors. Let N be the set of nodes in $S_t(Y, X)$ whose labels are known at the end of step t by at least one of these processors. By Lemma 1, each processor knows the label of at most 2^t nodes; therefore $|N| \leqq 2^{m+2t}$. Since no processors in $L_t(Y, X)$ are in $Q_t(Y, X)$, they do not know the outcome of any query in $U_t(Y, X)$, so changing the outcomes of some of these queries cannot change the set N.

When the outcomes of the queries in $U_t(Y, X)$ are allowed to vary, the node $R_{t+1}(Y, X)$ is equally likely to be any one of the 2^k nodes of depth k in $S_t(Y, X)$. This is because no label is repeated along any path and the labels of the nodes in $S_t(Y, X)$ are chosen independently of the outcomes of the queries in $U_t(Y, X)$. At most $|N|$ of the subtrees of $S_t(Y, X)$ rooted at these 2^k nodes can contain elements of N. Thus the probability that $S_{t+1}(Y, X)$ contains some node in N is at most $|N|2^{-k} \leqq 2^{m+2t-k} \leqq 2^{-T}$. Hence 2^{-T} is an upper bound on the probability that there is a processor (in $L_t(Y, X)$) that knows the label of a node in $S_{t+1}(Y, X)$ at the end of step t and, at step $t+1$, reads from a processor in $Q_t(Y, X)$.

Next, we show the probability is also small that there is a processor in $Q_t(Y, X)$ which, at step $t+1$, reads from a processor (in $L_t(Y, X)$) that knows the label of a node in $S_{t+1}(Y, X)$ at the end of step t.

For any processor P, let $N_P(Y, X)$ be the set of nodes in $S_{t+1}(Y, X)$ whose labels are known by P immediately after step t. If $P \notin L_t(Y, X)$, then $N_P(Y, X) = \phi$. Furthermore, it follows from Lemma 1 that $|N_P(Y, X)| \leqq 2^t$. Let $Q \in Q_t(Y, X)$ and let P_Q be the processor that Q reads from at step $t+1$. Note that, since $Q_t(Y, X) \cap L_t(Y, X) = \phi$, processor Q does not know the label of any node in $S_t(Y, X)$, so changing the label

of any node in $S_t(Y, X)$ does not change Q's state and, hence, which processor Q reads from at step $t+1$.

We prove that, with probability less than $2^{-T}+2^{11-T}$, the subtree $S_{t+1}(Y, X)$ contains a node whose label is known by some processor in $L_t(Y, X)$ that was read by a processor in $Q_t(Y, X)$. Since there are no more than $2^{m+t} \leqq 2^{4T}$ processors in $Q_t(Y, X)$, it suffices to prove that for an arbitrary processor $Q \in Q_t(Y, X)$, the probability $S_{t+1}(Y, X)$ contains a node whose label is known by processor P_Q is less than $2^{-5T}+2^{11-5T}$.

On input (Y, X), the state of processor Q is determined by the outcomes of at most 2^t queries. For any other input in which these queries have the same outcomes, processor Q will also be in the same state. Thus we may partition the set of possible outcomes for the queries into those that give rise to the same state of Q. Since we may assume, without loss of generality, that processors do not forget information, each class of the partition can be specified by a set Z of at most 2^t queries and outcomes z for those queries. Then

$$\begin{aligned}&\operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi] \\ &\quad = \sum_{(Z,z)} \operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z] \operatorname{pr}[Z = z],\end{aligned}$$

where the sum is taken over pairs (Z, z), one for each class of the partition. Now $\sum_{(Z,z)} \operatorname{pr}[Z = z] = 1$, so it suffices to show that

$$\operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z] < 2^{-5T}+2^{11-5T}$$

for any pair (Z, z) that specifies a class of the partition.

We will show that for most labelings y of the decision tree, the nodes whose labels are known by processor P_Q are unlikely to be contained in $S_{t+1}(y, X)$, where the probability is taken over all inputs that satisfy $Z = z$. Note that the set of nodes whose labels are known by processor P_Q may be a function of the labels of the nodes in the decision tree.

A path in $S_t(Y, X)$ from $R_t(Y, X)$ to a node at depth k is said to be *constrained* if it contains at least 11 nodes labeled by variables in Z. Let E be the event that no path of length k, starting from $R_t(Y, X)$, is constrained. Then

$$\begin{aligned}&\operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z] \\ &\quad = \operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z \wedge \bar{E}] \operatorname{pr}[\bar{E}] \\ &\qquad + \operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z \wedge E] \operatorname{pr}[E] \\ &\quad \leqq \operatorname{pr}[\bar{E}] + \operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z \wedge E].\end{aligned}$$

The labels of the nodes on a path can be viewed as being selected without replacement from the set $U_t(Y, X)$. Thus the probability that a particular path is constrained is at most

$$\binom{k}{11}\left(\frac{|Z|}{|U_t(Y, X)|-k}\right)^{11} < \left(\frac{k|Z|}{|U_t(Y, X)|-k}\right)^{11}$$

and the probability $\operatorname{pr}[\bar{E}]$ that some path in the tree is constrained is less than

$$2^k\left(\frac{k|Z|}{|U_t(Y, X)|-k}\right)^{11} \leqq 2^k\left(\frac{k2^t}{2^m-k(t+1)}\right)^{11} \leqq 2^{-5T} \quad \text{for } T \geqq 5.$$

Now consider any labeling y of the decision tree that agrees with Y from the root to $R_t(Y, X)$ and in which no path of length k from $R_t(Y, X)$ is constrained. The probability that $R_{t+1}(y, X)$ is any particular node at level k of $S_t(y, X)$ is at most 2^{11-k}. Thus the probability that $S_{t+1}(y, X)$, the subtree of $S_t(y, X)$ rooted at $R_{t\geqq 1}(y, X)$, contains a node in $N_{P_Q}(y, X)$ is at most

$$2^{11-k}|N_{P_Q}(y, X)| \leqq 2^{11-k+t} \leqq 2^{11-5T}.$$

Hence, $\operatorname{pr}[N_{P_Q}(y, X) \cap S_{t+1}(y, X) \neq \phi \mid Z = z \wedge E] \leqq 2^{11-5T}$ and $\operatorname{pr}[N_{P_Q}(Y, X) \cap S_{t+1}(Y, X) \neq \phi \mid Z = z] \leqq 2^{-5T} + 2^{11-5T}$, as desired.

Combining the information about both cases, we get that

$$\begin{aligned}&\operatorname{pr}[Q_{t+1}(Y, X) \cap L_{t+1}(Y, X) \neq \phi \mid Q_t(Y, X) \cap L_t(Y, X) = \phi] \\ &\quad < 2^{-T} + 2^{-T} + 2^{11-T} < 2^{12-T}\end{aligned}$$

and

$$\begin{aligned}&\operatorname{pr}[Q_{t+1}(Y, X) \cap L_{t+1}(Y, X) \neq \phi] \\ &\quad \leqq \operatorname{pr}[Q_t(Y, X) \cap L_t(Y, X) \neq \phi] \\ &\qquad + \operatorname{pr}[Q_{t+1}(Y, X) \cap L_{t+1}(Y, X) \neq \phi \mid Q_t(Y, X) \cap L_t(Y, X) = \phi] \\ &\quad < t2^{12-T} + 2^{12-T} = (t+1)2^{12-T}.\end{aligned}$$

Thus the claim is true.

5. Conclusions. This paper shows that there is a Boolean function of n variables that can be computed by a CROW PRAM in $O(\log \log n)$ steps, but any EROW PRAM that computes it has expected running time in $\Omega(\sqrt{\log n})$. We think that there is a similar separation between CROW PRAMS and EREW PRAMs. Note that, for computing Boolean functions, CROW PRAMs are as powerful as CREW PRAMs (to within a constant factor) and, hence, are at least as powerful as EREW PRAMs. However, this is not necessarily the case if the domain is not complete. (For example, consider the OR of n Boolean values, at most one of which is 1.)

There is a very close correspondence between CROW PRAMs and decision trees [4]. If a function (over any domain) can be computed by a CROW PRAM in time T, then it can be computed by a decision tree of height 2^T. (In particular, this implies that any function computable in constant time on a CROW PRAM is also computable in constant time on a PRAM with only one processor.) Conversely, if a function can be computed by a decision tree of height h, then it can be computed by a CROW PRAM in time $\lceil \log_2 h \rceil + 1$. Thus the Boolean decision-tree evaluation problem is complete for CROW PRAM computation in the following sense. If a function $f: \{0, 1\}^n \to R$ can be computed by a CROW PRAM in time $t(n)$, then f can be computed by an EREW PRAM (or any other model of computation) using no more time than it takes to compute $D_{n,2^{t(n)}}$.

We conjecture that, on the EREW PRAM, a $(\log n)^{\Omega(1)}$ lower bound can be obtained for the Boolean decision-tree evaluation problem for appropriate choices of m and h. Unfortunately, the proof of Theorem 3 does not appear to generalize in a straightforward way. The essential problem is that, on CREW and EREW PRAMs, information about some input bits can be transmitted to a memory cell by virtue of the fact that no value was written there during a particular step of a computation. (See [2] for details.) The definition of knowledge must be modified to take this into account.

For their CREW PRAM lower bound, Cook, Dwork, and Reischuk [2] used the following definition to capture certain properties of knowledge. A processor or memory cell is said to be *affected* by a particular input bit at time t on input x if the state of the processor or the contents of the memory cell immediately after step t of the computation is different for x than for the input obtained from x by changing the value of the specified input bit. This definition supports lemmas analogous to Lemmas 1 and 2, provided the input domain is assumed to be $\{0, 1\}^n$.

LEMMA 4 (Cook, Dwork, and Reischuk [2]). *For a* CREW PRAM, *on any input, every processor and memory cell is affected by at most* $(\frac{1}{2}(5+\sqrt{21}))^t$ *input bits immediately after step t.*

LEMMA 5 (Beame [1]). *For an* EREW PRAM, *on any input, each input bit affects at most* $(2+\sqrt{3})^t$ *processors and memory cells immediately after step t.*

There is another fact about knowledge used in the proof of Theorem 3 that, unfortunately, is not shared by the affects relation. If a processor or memory cell does not know certain input bits, then changing all of their values does not change the state of the processor or the contents of the memory cell. Moreover, the set of input bits that the processor or memory cell knows remains unchanged.

This motivates the following definition. A set of input bits is a *dependency set* for a processor or memory cell at time t on input x if the state of the processor or the contents of the memory cell immediately after step t of the computation is the same for x as it is for the inputs obtained from x by changing the values of any set of bits not in the dependency set. Furthermore, there is another version of Lemma 1 that holds for the CREW PRAM with the complete input domain $\{0, 1\}^n$.

LEMMA 6 (Nisan [7]). *For a* CREW PRAM, *on any input, every processor and memory cell as a dependency set containing at most* $(\frac{1}{2}(5+\sqrt{21}))^{2^t}$ *input bits immediately after step t.*

Note that this lemma does not imply that, after a small number of steps, all (minimal) dependency sets are small. For example, consider the contents of the output cell at the end of the computation of $D_{m,1}: \{0, 1\}^{m+2^m} \to \{0, 1\}$. Recall that for $y \in \{0, 1\}^m$ and $x_0, \cdots, x_{2^m-1} \in \{0, 1\}$,

$$D_{m,1}(y, x_0, \cdots, x_{2^m-1}) = x_y$$

and this function can be computed in $O(\log m)$ steps. On input 0^{m+2^m}, the last 2^m bits comprise a minimal dependency set.

Even if we could somehow associate a small dependency set with each processor and memory cell, the corresponding version of Lemma 2 would also be false. Suppose, for example, that during the first $O(t)$ steps of a computation, a processor accumulates the values of 2^t input bits and then writes a special value to the memory indexed by this 2^t-tuple. Each of the 2^{2^t} memory cells in which the special value can appear must have at least one of these 2^t input bits in its associated dependency set. Hence, at least one of these input bits must be a member of the dependency sets associated with at least 2^{2^t-t} different shared memory cells.

The notions of affects and dependency set do not suffice to extend the proof of Theorem 3. However, understanding these and other related definitions will provide us with additional insight into the nature of exclusive read. We also believe that a definition of knowledge can be obtained to show the Boolean decision-tree evaluation problem is hard on EREW PRAMs.

The CROW PRAM model can be extended by allowing each processor to own many different shared memory cells, instead of just one. At each timestep, a processor could write to any one of the memory cells it owns. However, each memory cell would

still be owned by only one processor. This new model is no more powerful than the CROW PRAM because each CROW PRAM processor could use its single shared memory cell to record the entire sequence of values that would have been written and the locations to which they would have been written. All interested processors could then read this information.

The EROW PRAM model can be extended in the same way. But it is not clear whether the resulting model is more powerful than the EROW PRAM and whether it is less powerful than the EREW or CROW PRAMs. One approach to obtaining our desired separation between EREW and CROW PRAMs is to first attempt to prove that the Boolean decision-tree evaluation problem is hard on this model.

REFERENCES

[1] P. BEAME, *Lower bounds in parallel machine computation*, Tech. Report 198/87, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1987.

[2] S. COOK, C. DWORK, AND R. REISCHUK, *Upper and lower time bounds for parallel random access machines without simultaneous writes*, SIAM J. Comput., 15 (1986), pp. 87–98.

[3] P. DYMOND AND W. L. RUZZO, *Parallel* RAMs *with owned global memory and deterministic context-free language recognition*, in Proc. 13th International Colloquium on Automata, Languages and Programming, 1986, pp. 95–104.

[4] F. FICH AND P. RAGDE, unpublished manuscript.

[5] E. GAFNI, J. NAOR, AND P. RAGDE, *On separating the* EREW *and* CROW *models*, Theoret. Comput. Sci. to appear.

[6] R. KARP AND V. RAMACHANDRAN, *A survey of parallel algorithms for shared-memory machines*, Tech. Report UCB/CSD 88/408, University of California, Berkeley, CA, 1988.

[7] N. NISAN, CREW PRAMs *and decision trees*, in Proc. 21st Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1989, pp. 327–335.

[8] M. SNIR, *On parallel searching*, SIAM J. Comput., 14 (1985), pp. 688–708.

[9] A. YAO, *Probabilistic computations: toward a unified measure of complexity*, in Proc. 18th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1977, pp. 222–227.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 728-741, August 1990

012

SCHEDULING SEQUENTIAL LOOPS ON PARALLEL PROCESSORS*

ASHFAQ A. MUNSHI† AND BARBARA SIMONS‡

Abstract. Automatic parallelization of code written in a sequential language such as FORTRAN is of great importance for compilers for parallel computers. First, the problem of automatically parallelizing iterative loops on multiprocessors is discussed, and then a scheduling problem involving precedence constraints that models a technique for the automatic parallelization is derived. Polynomial time algorithms are presented for some special cases of this scheduling problem together with an upper bound on a naive algorithm for the general case. Using one of the polynomial time algorithms, a heuristic for the original compiler problem is obtained. Finally, test results obtained by applying our heuristic to EISPACK, a well-known numerical analysis FORTRAN package, are presented. In these tests the amount of parallelism obtained always equals and frequently surpasses that obtained by the best known techniques in the literature. This approach represents one of the first attempts at understanding the complexity theoretic aspects of loop parallelization.

Key words. compiler, parallel computing, sequential code, loop, scheduling, dependences, precedence constraints

AMS(MOS) subject classifications. primary 68N20; secondary 68Q25, 68R10

1. Introduction.

While much research and development is being devoted to designing parallel machines and parallel algorithms, there is a large amount of code that has been written for sequential machines in languages such as FORTRAN. It is expensive and time-consuming to rewrite this code so that it will run efficiently on a parallel machine. To further complicate the problem, people frequently do not know precisely what the sequential code does or what side effects it might have. Consequently, there has been considerable work on the problem of generating parallel code from sequential code [KKLW]. Much of this work has focused on detecting parallelism in loops, especially FORTRAN DO loops [Lam]. One important technique, known as DOACROSS [Cyt], assigns a unique processor to execute the loop for a unique value of the loop iteration variable. That is, if a loop is to be executed N times, and if there are at least N processors, then one processor is assigned to each iteration of the loop. Ideally, all N processors start at the same time and consequently finish at approximately the same time. Because of data dependences within the loop, however, this idealized version of DOACROSS usually cannot be realized. Data dependences may force certain processors to wait for results that must be computed by other processors before they can continue their local computations.

More precisely, let K be the iteration variable of the loop. Suppose that each loop iteration is run on a different processor, and assume that each statement has unit execution time. Label the statements in the loop in the order in which they initially occur, i.e., $s_1, s_2, \cdots, s_n$. Suppose that statement s_j accesses a memory location in iteration K that is accessed by statement s_i in iteration $K-1$. Then we say that s_j *depends* on s_i. If $i > j$, then processor K must wait until processor $K-1$ has computed s_i before processor K can compute s_j. If $i < j$, then no delay is necessary, since by the time processor K reaches s_i, processor $K-1$ has already computed s_j. Finally, if $i = j$, then the delay must be at least 1, since processor K must wait for statement i to be

* Received by the editors September 27, 1987; accepted for publication (in revised form) October 13, 1989.
† Oracle Corporation, 20 Davis Drive, Belmont, California 94002. This work was done while the author was at IBM Almaden Research Center, San Jose, California.
‡ IBM Almaden Research Center, K53-802, 650 Harry Rd., San Jose, California 95120-6099.

executed by processor $K-1$. The length of a data dependence is $|j-i|$, and it is said to be *lexicographically backward* if $j-1 \leqq 0$.

A simple extension of the above discussion shows that the amount of delay is precisely the length of the longest backward dependence plus one [Cyt].[1] For example, if there are no backward dependences, the delay is zero. A dependence from a statement to itself (i.e., its occurrence in the previous iteration) has length zero and delay one. If there is a backward dependence extending from the last statement of the loop to the first statement of the loop, the length is $n-1$, and the delay is n, the number of statements in the loop. A loop is "parallel" if it has zero delay and "sequential" if the delay is equal to the number of statements in the loop.

A natural question is how to minimize the delay associated with a loop by rearranging the order of the code. This question is of considerable practical importance, since its solution would allow a compiler to run sequential programs more efficiently on parallel architectures.

2. Preliminaries and definitions. Most programs have two types of dependences: data dependence and control dependence. Control dependences are caused by conditional and unconditional branching. It has been shown [AK], [AKPW], [All] that control dependences can be turned into data dependences via some simple transformations on the original program. The new program is semantically equivalent to the original one but possesses only data dependences.

An alternative approach for dealing with control dependences is to eliminate unnecessary control dependences by constructing a Program Dependence Graph (PDG). (See [SAF] for an in-depth discussion of PDGs.) The control dependences of the PDG are treated by our algorithm as if they were loop independent data dependences (defined below).

We assume that control dependences have been dealt with in some manner. A data dependence is defined as follows. Statement s_i is said to be *data dependent* on statement s_j when the following two conditions hold:

(1) There exists a possible execution path such that statements s_j and s_i both reference the same memory location M, and

(2) the execution of s_j that references M occurs before the execution of s_i that references M [All].

Data dependence can be separated into two types: *loop independent* dependences and *loop carried* dependences. A loop independent dependence of s_i on s_j exists if s_i depends on s_j for a fixed value of the loop iteration variable; otherwise, the dependence is said to be loop carried. Intuitively, a loop carried dependence requires the presence of the loop around the body of statements, whereas a loop independent dependence is present regardless of the loop. If two statements participate in a loop carried dependence, they can be permuted at will; statements in a loop independent dependence must occur in the given order.

Loops may be nested, or loop carried dependences might arise because of dependences on data computed several iterations prior to the current iteration. These potential complications are eliminated as follows. We define *distance* to be the number of iterations one has to go back to find the source of the dependence. If the distance is greater than one, then the problem can be reduced to the distance one case by *unrolling*

[1] Alliant Computer Corporation uses the DOACROSS technique of [Cyt] for running sequential FORTRAN programs on its eight processor machine, the FX/8.

the loop. That is, if the distance is $k > 1$, then each successive set of k iterations of the loop is viewed as a single "large" loop. For nested loops the innermost loops are first processed as if they are isolated loops, that is, as if they are not nested. Each innermost loop is then collapsed into a single statement, and the process is repeated on the new innermost loops until finally there is only one loop remaining.

The above simplifications allow us to assume that dependences occur only as a result of the previous iteration. We also assume that there are no nested loops. (For a further discussion see [Cyt] and [Mun].) Finally, we assume that the statements all have equal execution time.

Given a loop L, the *dependence graph*, $G_L(N, A)$, of L is a directed graph where the nodes of G_L represent statements and the arcs represent data (and control) dependences. Figure 2 is the dependence graph for the loop of Fig. 1.

As an example, consider the data dependences of the DO loop in Fig. 1. Note that s_4 depends on s_3 because s_4 references memory location A3(I) after s_3. Furthermore, the dependence is a loop independent dependence, for it occurs no matter what value I assumes. By contrast, statement s_2 has a loop carried dependence on s_8 because s_2 and s_8 both reference memory location A8(I), but the reference is for different values of I. Similarly, s_1 has a loop carried dependence on itself since A1(I−1) is needed to compute A(I). This dependence is represented as a "self loop" and implies that the overall delay must be at least one. Since it is not possible to eliminate the delay of one caused by self loops, we ignore them in the construction of the scheduling problem. If, subsequently, the heuristic succeeds in rearranging the statements of the loop to

```
SUBROUTINE SAMPLE
(A1, A2, A3, A4, A5, A6, A7, A8, A9, A10, A11, A12, A13, A14, A15, A16, A17, N)
REAL A1(N), A2(N), A3(N), A4(N), A5(N), A6(N), A7(N), A8(N), A9(N), A10(N), A11(N), A12(N),
A13(N), A14(N), A15(N), A16(N), A17(N)
```

	Loop independent			Loop carried
	DO 227 I = 1, N			
s_1	A1(I) =			A1(I−1)
s_2	A2(I) =			A8(I−1)
s_3	A3(I) =			A5(I−1)
s_4	A4(I) =	A3(I)	+	A7(I−1)
s_5	A5(I) =	A2(I)		
s_6	A6(I) =	A1(I)	−	A13(I−1)
s_7	A7(I) =	A4(I)		
s_8	A8(I) =	A4(I) + A5(I)	+	A17(I−1)
s_9	A9(I) =	A1(I)		
s_{10}	A10(I) =	A9(I)	*	A15(I−1)
s_{11}	A11(I) =	A9(I)		
s_{12}	A12(I) =	A9(I)		
s_{13}	A13(I)	A12(I)	−	A9(I−1)
s_{14}	A14(I) =	A13(I)		
s_{15}	A15(I) =	A14(I)		
s_{16}	A16(I) =	A14(I)		
s_{17}	A17(I) =	A14(I)		
227	CONTINUE			
	RETURN			
	END			

FIG. 1. *A* FORTRAN DO *loop.*

allow for a delay of zero, we impose a delay of one to satisfy the constraint of the self loop. Applying the above definitions gives the dependence graph of Fig. 2.

Since control and loop independent dependences must be acyclic, any backwards dependence arcs must be loop carried data dependences. Therefore, an optimal schedule, which can be represented as a permutation of the statements of the original schedule, either eliminates all backward arcs or minimizes the length of any backward arcs that cannot be eliminated. The delay optimization problem can now be phrased succinctly:

Given a loop L, find a permutation π of the dependence graph $G_L(N, A)$ such that the semantics of the loop are preserved and $\max_{(s_i,s_j)\in G_L(A)}\{\pi(s_i)-\pi(s_j)\}$ is minimized over all backward arcs.

As was mentioned above, two statements participating in a loop carried dependence can be permuted while preserving the semantics of the loop. Therefore, a first approximation to the delay problem is to consider a loop that contains only loop carried dependences, i.e., every permutation of the statements preserves the semantics. Unfortunately, even determining if there is a permutation of the statements such that the delay is no greater than some value λ is NP-complete for this simple problem [Cyt]. The reduction is from bandwidth minimization.[2]

In general, it is not possible to make arbitrary permutations of the statements within the loop, since control and loop independent data dependences force a partial order on the statements of the loop. Note that if the dependence graph does not have any circuits, then a topological sort of the graph results in no backward arcs, which in turn implies no delay.

We view the problem of minimizing delay as having two parts. The first part is the problem of mapping the FORTRAN DO loop to an instance of a scheduling problem, and the second part is the solution of the scheduling problem.

3. Constructing the scheduling problem. Given a loop L, let $G_{LI}(N, A)$ be the dependence graph of L that is obtained when the dependences are restricted to control and loop independent data dependences. G_{LI} is partitioned into *levels* as follows. All

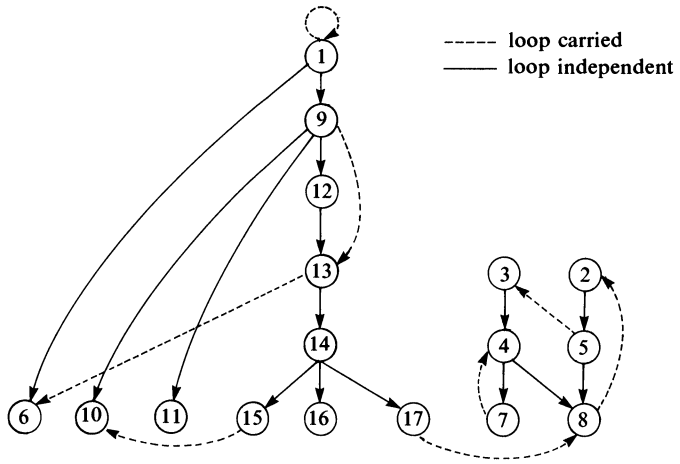


FIG. 2. *The dependence graph for Fig.* 1.

[2] Anderson, Munshi, and Simons [AMS] have shown that the Delay problem is NP-complete for a forest of arbitrary trees (i.e., the trees are neither in-trees nor out-trees).

the nodes of G_{LI} that have no successors (i.e., sinks) are at level 1. In general, all the nodes of G_{LI} that have successors only at level $i-1$ and possibly at levels below level $i-1$ are at level i. Note that if two nodes are at the same level, then they are independent with respect to G_{LI}. Therefore, G_{LI} can be partitioned into a set of independent sets $\Sigma = \{\Sigma_i\}$ by assigning all nodes at level i to the set Σ_i. We define the obvious total ordering $<$ on $\Sigma: \Sigma_i < \Sigma_j$ if and only if $i < j$.

Even though the nodes in Σ_i form an independent set in G_{LI}, they might contain some loop carried dependences. If there are loop carried dependences among the nodes in Σ_i, then Σ_i is partitioned into subsets until no loops carried dependences remain among the nodes in each of the subsets. A total ordering is then applied to all the subsets and any remaining unpartitioned sets such that the total ordering among the original sets in Σ is not violated.

An arc (s_i, s_j) with $s_i \in \Sigma_p$, $s_j \in \Sigma_q$ is *forward* if $p > q$ and *backward* otherwise. (The sets Σ_i have been constructed so that all control and independent data dependences go from higher numbered Σ sets to lower numbered ones. Therefore, the backward loop carried data dependences whose lengths we want to minimize all go from lower numbered Σ sets to higher numbered ones.) If the loop carried dependences of Σ_i do not form a loop in Σ_i, then the subsets of Σ_i can be ordered such that none of the loop carried dependences is a backward arc. However, for some cases such an ordering of the subsets will be globally suboptimal.[3]

Assume we have a set of independent sets of nodes $\Sigma_1, \Sigma_2, \cdots, \Sigma_L$. We assign sets of integers to the independent sets in Σ as follows. The integers from 1 to $|\Sigma_L|$ are assigned to Σ_L, the integers from $|\Sigma_L|+1$ to $|\Sigma_L|+|\Sigma_{L-1}|$ are assigned to Σ_{L-1}, and so on. The above construction partitions the integers $1 \cdots n$, where n is the number of statements in the loop (as well as the number of nodes in G_{LI}). Furthermore, the mapping of the nodes onto the integers reflects the ordering imposed between the Σ sets by the control and loop independent data dependences.

In the example of Fig. 3, the first Σ set constructed from only the control and loop independent dependences is {6, 7, 8, 10, 11, 15, 16, 17}. Although these nodes are independent with respect to the control and loop independent data dependences, there are loop carried dependences from 15 to 10 and from 17 to 8. Therefore, the set has to be partitioned into two Σ sets, each of which contains one node from each of the loop carried dependences. Figure 3 shows a legal partitioning of the nodes and the number line for Fig. 1 with $\Sigma_1 = \{15, 17\}$, $\Sigma_2 = \{6, 7, 8, 10, 11, 16\}$, $\Sigma_3 = \{4, 5, 14\}$, $\Sigma_4 = \{2, 3, 13\}$, $\Sigma_5 = \{12\}$, $\Sigma_6 = \{9\}$, $\Sigma_7 = \{1\}$.

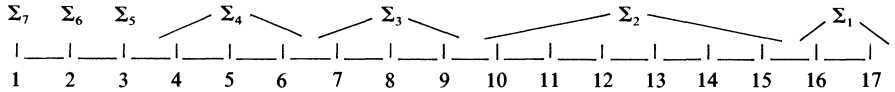


FIG. 3. *Independent sets and a partition of the number line for Fig.* 1.

In addition to mapping nodes to partitions of the first n integers, we construct a DAG that we call $B(N, A)$, where $N(B) = N(G_L)$ and $A(B)$ consists of all arcs $(s_i, s_j) \in A(G_L)$ for which j is assigned to an earlier partition than is i. The scheduling problem consists of assigning an integer start time to all the nodes in $N(B)$ such that

[3] While our techniques for constructing the independent sets Σ_i seem to work well in practice, an implementor might wish to experiment with variations.

(1) Each node is *scheduled*, that is, assigned a start time, within the interval to which its set is assigned.

(2) The start times of any two nodes differs by at least one.

(3) The difference in start times of the endpoints of any arc in $A(B)$ is minimized. More formally:

The delay problem. Given an ordered set of disjoint intervals $\mathcal{I} = I_1, \cdots, I_k$, a DAG $B(N, A)$ of precedence constraints, and a mapping $f: N(B) \to \mathcal{I}$ so that if $(i, j) \in A(B)$, then $f(i) > f(j)$, find a mapping $s: N(B) \to I^+$ such that for all $i, j \in N(B)$, $s(i) \in f(i)$, $s(j) \in f(j)$, $s(i) - s(j) \geqq 1$ for $i \neq j$, and $\max_{(i,j)\in A(B)} \{s(i) - s(j)\}$ is minimized.

To illustrate, consider again the dependence graph of Fig. 2 in which loop independent data dependences force s_1 to precede s_9, which in turn must precede s_{12}. Hence, the set containing s_1 must precede the set containing s_9, which must precede the set containing s_{12}. Since the arc representing the loop carried dependence $s_{13} \to s_6$ is no longer a backward arc in the partition given in Fig. 3, the set of backward arcs involves only the statements $s_2, s_3, s_4, s_5, s_7, s_8, s_{10}, s_{15}$, and s_{17}. Consequently, $B(N, A)$ contains only nodes corresponding to these statements and only those arcs that are backward arcs. The nodes corresponding to statements s_2 and s_3 must be scheduled in the interval [4, 6], the nodes corresponding to s_4 and s_5 in [7, 9], the nodes corresponding to s_7, s_8, and s_{10} in [10, 15], and the nodes corresponding to s_{15} and s_{17} in [16, 17].

This completes the mapping to the scheduling problem. For ease of exposition we assume for the remainder of the paper that all arcs in the scheduling problem go forward. So, if (s_i, s_j) is an arc, then the interval into which node i is mapped comes before the interval into which node j is mapped on the number line. The reader should understand that an arc in a graph for the scheduling problem is actually a backward arc from the corresponding dependence graph.

4. Scheduling chains.[4] A *chain* $C = B(N, A)$ is a simple path where $N(B) = \{C_1, C_2, \cdots, C_n\}$ with $A(B) = \{(C_i, C_{i+1})\}$. We first consider the version of the Delay Problem in which $B(N, A)$ is a single chain.

Problem SC (single chain). Let $C = B(N, A)$ be a single chain with $f(C_i) = I_i$. Let the left and right endpoints of these (closed) intervals be l_i and r_i, respectively, with $1_i \leqq r_i < l_{i+1} \leqq r_{i+1}$ for $1 \leqq i < n$, and r_i and l_i integers for $1 \leqq i \leqq n$. Find a mapping $s: N(B) \to I^+$ such that $s(C_i) \in I_i$ and $\max_{i<n} (s(C_{i+1}) - s(C_i))$ is minimized.

We model the SC problem as a scheduling problem. A *schedule* is an assignment of start times to each node in the problem instance. A *feasible schedule* is a schedule in which each node is assigned a unique start time that falls within its assigned interval and in which the difference between any two start times is at least one. If the integer $c \in f(i)$, then c is called a *slot*. If no element of $N(B)$ is mapped to c, then c is said to be *empty*.

An approach to solving problem SC is to initially schedule each node C_i at the rightmost slot of its interval r_i. Then a left shift y_i is computed for each C_i so that the maximum separation between adjacent nodes is minimized.

---

[4] All the algorithms presented in this paper can be easily modified to work for the case in which the endpoints are rational. We use integer values throughout because of the formulation of the original compiler problem.

Mathematically, this amounts to solving the following optimization problem:

$$\min_{y_i} \max_{i} \{(r_{i+1} - y_{i+1}) - (r_i - y_i)\}$$
$$\text{subject to } 0 \leqq y_i \leqq r_i - l_i,\ l_i,\ r_i, \text{ and } y_i \text{ integers.}$$

This is equivalent to solving the integer linear program

$$\min \lambda$$
$$\text{subject to } (r_{i+1} - y_{i+1}) - (r_i - y_i) \leqq \lambda$$
$$\text{and}$$
$$0 \leqq y_i \leqq r_i - l_i,\ r_i,\ \lambda \text{ integers.}$$

If inequalities i through $i+j$ are added together, we get $(r_{i+j} - y_{i+j}) - (r_i - y_i) \leqq j\lambda$. Since $0 \leqq y_{i+j} \leqq r_{i+j} - l_{i+j}$ we have $y_{i+j} - y_i \leqq r_{i+j} - l_{i+j}$. Adding this to the previous inequality gives

$$\lambda \geqq \lceil ((l_{i+j} - r_i)/j) \rceil = \lambda_{ij} (\dagger).$$

This implies that $\lambda \geqq \max_{i,j} \{\lambda_{ij}\}$, since i and j were chosen arbitrarily. We now show that this lower bound is tight.

Let $s(C_i)$ be the starting time of C_i, where C_i started at $s(C_i)$ executes in the interval $[s(C_i), s(C_i)+1)$. The algorithm for achieving the lower bound (†) is

Algorithm SC (λ)

Set $s(C_1) = r_1$.
Set $s(C_i) = \min (s(C_{i-1}) + \lambda, r_i)$, where $\lambda = \max_{ij} \{\lambda_{ij}\}$, for $i = 2, \cdots, n$.

To prove that algorithm SC is correct, we assume for contradiction that for some node C_p, $s(C_p) < l_p$. Let $q = \max \{_{k<p}(s(C_k) = r_k)\}$. There must exist such a q since $s(C_1) = r_1$. Therefore, $r_q + (p-q)\lambda < l_p$, which implies $\lambda < (l_p - r_q)/(p-q)$, a contradiction. This proves the following theorem.

THEOREM 1. *Algorithm* SC *is an $O(n^2)$ time algorithm for optimally scheduling the nodes of a single chain, where n is the number of nodes.*

Because the SC problem is symmetric, the following algorithm also solves SC.

Algorithm REVERSE-SC (λ)

Set $s(C_n) = l_n$
Set $s(C_j) = \max (s(C_{j+1}) - \lambda, l_j)$ for $j = 1, \cdots, n-1$.

Yet another algorithm for the SC problem is obtained by performing binary search on λ and checking feasibility using algorithm SC with the test value of λ. For the original compiler problem $\lambda \leqq n$; for this case the binary search approach gives an algorithm with a running time of $O(n \log n)$. In summary,

LEMMA 2. *There exists an $O(n \log \lambda)$ algorithm for Problem* SC.

We next consider the problem of scheduling an arbitrary number of chains.

Let $C = \{C_1, C_2, \cdots, C_n\}$ be a set of chains, with C_i being the simple path $C_{i,1}$, $C_{i,2}, \cdots, C_{i,n_i}$, and let $\mathcal{I} = \{I_1, I_2, \cdots, I_m\}$ be a set of intervals, with $1_i \leqq r_i < l_{i+1} \leqq r_{i+1}$, for $1 \leqq i < m$, and r_i, l_i integers for $1 \leqq i \leqq m$. The *intervals associated with* C_i are the subset of $\mathcal{I}$ to which the nodes of C_i are mapped by f. The *underlying* interval of a node is the interval into which the node is mapped by f. A λ-feasible schedule is a feasible schedule in which the difference between the start times of two nodes connected by an arc is no greater than λ.

Problem Multiple Chain (MC). Given a set of intervals $\mathcal{I} = I_1, \cdots, I_m$ with $l_i \leqq r_i < l_{i+1} \leqq r_{i+1}$, for $1 \leqq i < m$, and a set of n chains $C_1, \cdots, C_n$, compute the minimum value

of λ such that there exists a λ-feasible schedule for the problem instance. Algorithm MC (λ), presented below, constructs a feasible schedule if one exists for an instance of problem MC and a given value of λ. If no feasible schedule exists, it determines that there is none for the given value of λ.

The *λ-feasible intervals* for a chain C_i are the intervals obtained by applying algorithm SC (λ) and REVERSE-SC (λ) for a fixed value of λ to C_i together with the intervals associated with C_i. The left endpoint of the λ-feasible interval for $C_{i,j}$ is the start time assigned to $C_{i,j}$ by REVERSE-SC (λ), and the right endpoint is the start time assigned by SC (λ). The λ-feasible interval for $C_{i,j}$ is denoted by λ-*feas*$(C_{i,j})$, with the left endpoint (right endpoint) of λ-feas $(C_{i,j})$ being denoted *left*(λ-*feas*$(C_{i,j})$) (right (λ-feas $(C_{i,j})$)). We drop the use of λ when the meaning is obvious from the context. The left (right) endpoint of an interval I_k is denoted by *left*(I_k) (*right*(I_k)). Given a schedule (which may not be a feasible schedule) for an instance of MC, a *conflicting set* is a set of two or more nodes which have the same start time. The rightmost conflicting set is the conflicting set with the latest start time.

Algorithm MC (λ)

(1) Compute the λ-feasible intervals of each chain C_i separately. If SC (λ) does not construct a feasible schedule for some chain, then declare the problem instance infeasible and halt.

(2) Schedule each node at the rightmost slot of its feasible interval.

(3) If each node is assigned a unique start time, then output the schedule and halt.

(4) Let S be the rightmost conflicting set and let $S' \subseteq S$ be the set of nodes in S that are scheduled at the left endpoint of their feasible interval. If $|S'| \geqq 2$, then declare the problem instance infeasible and halt.

4(a) If there is a node $x \in S - S'$ such that x has no successor, shift x left one unit else

4(b) Let x be the node in $S - S'$ whose successor is scheduled earliest among all successors of nodes in $S - S'$. Shift x left 1 unit.

Go to step 3.

In the following example the chains are characterized by the underlying intervals into which the chain nodes are assigned.

Example. $\mathcal{I} = \{I_j : 1 \leqq j \leqq 6\}$, with $I_1 = [1 \cdots 4]$, $I_2 = [5 \cdots 7]$, $I_3 = (8 \cdots 12]$, $I_4 = [13 \cdots 15]$, $I_5 = [16 \cdots 21]$, $I_6 = [22 \cdots 24]$.

$\mathcal{C} = \{A, B, C, D, E, F\}$, with $A = [I_1, I_3, I_4, I_5, I_6]$, $B = [I_1, I_3, I_5]$, $C = [I_1, I_2, I_4, I_5]$, $D = [I_1, I_2, I_3, I_5, I_6]$, $E = [I_2, I_3, I_5]$, $F = [I_3, I_4, I_5, I_6]$.

Observe that $\lambda = 5$ is an infeasible value for the B chain, since $\lambda = 5$ implies that the third node in the B chain cannot start sufficiently late to reach I_5. However, for $\lambda = 6$, none of the chains is infeasible, and we get the following 6-feasible intervals:

$$A = [2, 4], [8, 10], [13, 15], [16, 21], [22, 24]$$
$$B = [4, 4], [10, 10], [16, 16]$$
$$C = [1, 4], [7, 7], [13, 13], [16, 19]$$
$$D = [1, 4], [5, 7], [10, 12], [16, 18]$$
$$E = [5, 7], [10, 12], [16, 18], [22, 24]$$
$$F = [8, 12], [13, 15], [16, 21], [22, 24]$$

The initial schedule in which all nodes are scheduled at the rightmost slots of their feasible intervals is illustrated in Fig. 4; Fig. 5 shows the final 6-feasible schedule.

Let SCH_k be the schedule constructed by Algorithm MC after $k \geqq 0$ left shifts.

LEMMA 3. *Suppose that for any x, $x' \in SCH_k$ such that x is the parent of x' we have $s(x') - s(x) \leqq \lambda$. Then, in SCH_{k+1} $s(x') - s(x) \leqq \lambda$.*

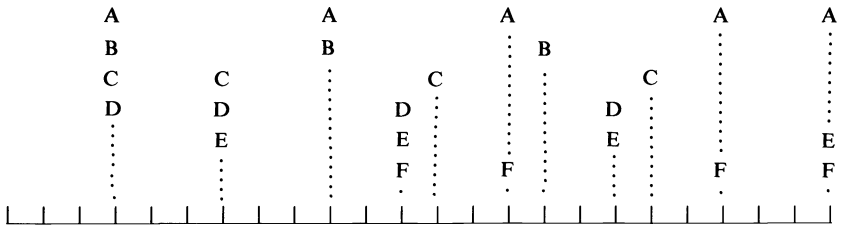


FIG. 4. *The initial placement of nodes for* $\lambda = 6$.

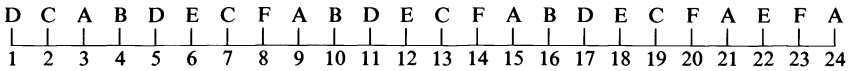


FIG. 5. *A 6-feasible schedule for the example of Fig.* 4.

Proof. The lemma clearly holds for all x, x' such that x is not the node that is shifted left on the $k+1$st left shift. Suppose that x is the node that is shifted left on the $k+1$st left shift. If x is selected by step 4(a), then the lemma follows trivially. So suppose x is selected by step 4(b). Then if x is not the only node in $S-S'$, there exists a $y \in S-S'$ with immediate successor y' such that $s(y') > s(x')$ in SCH_k. In SCH_{k+1} we have $s(y')-s(y) \leqq \lambda$, $s(y') \geqq s(x')+1$, and $s(x)+1 = s(y)$. Therefore, $s(x')-s(x) \leqq \lambda$ in SCH_{k+1}. Finally, suppose that $S-S' = \{x\}$ in SCH_k. Then, since $|S| \geqq 2$, there exists some node $y \in S'$, which implies that $s(y) = \text{left (feas } (y))$ in SCH_k. Since both x and y are mapped to the same underlying interval, I_j, and since left (feas $(y)) >$ left (feas (x)), y has an immediate successor y'. Suppose that $s(x')-s(x) = \lambda$ in SCH_k. By the induction assumption $s(y')-s(y) \leqq \lambda$. Because $s(y) = \text{left (feas } (y))$, $s(y')-s(y) = \lambda$. But this implies that $s(y') = s(x')$ in SCH_k, a contradiction since S is the leftmost conflicting set. Therefore, $s(x')-s(x) < \lambda$ in SCH_k; hence, $s(x')-s(x) \leqq \lambda$ in SCH_{k+1}. □

LEMMA 4. *If Algorithm* MC (λ) *has performed k left shifts, then for all x, x' such that x immediately precedes x',* $s(x')-s(x) \leqq \lambda$ *in* SCH_k.

Proof. The basis follows trivially from the construction of feasible intervals together with step 2 of Algorithm MC, and the induction step follows from Lemma 3. □

THEOREM 5. *If Algorithm* MC (λ) *halts at step* 3, *then it as constructed a* λ*-feasible schedule.*

Proof. Since all nodes are scheduled within their λ-feasible intervals, every node is scheduled within its underlying interval. Since all conflicting sets are eliminated, no two nodes have the same start time. Lemma 4 guarantees that any pair of nodes that are adjacent in some chain are scheduled no more than λ apart. □

LEMMA 6. *At the beginning of an iteration of step* 4, *let S be the rightmost conflicting set and let* $x \in S$ *be the node that is selected by the iteration of step* 4 *to be shifted left. Then left* (*feas*$(x)) \leqq$ *left* (*feas*(y)) *for all* $y \in S$.

Proof. If x is shifted left by step 4(a), then since x has no successors it follows from the definition of λ-feasible intervals that left (feas (x)) is also the left endpoint of the underlying interval. Hence the lemma holds.

Suppose that x is shifted left by step 4(b). Let y be any other element of S. If $y \in S'$, then left (feas $(x)) \leqq$ left (feas (y)). So suppose that $y \notin S'$ and let x' and y' be the successors of x and y respectively. Note that both x' and y' must exist, since otherwise either x or y would have been shifted left by step 4(a). Also, since S is the rightmost conflicting set, $s(x') \neq s(y')$.

We use an induction argument based on the number of shifts.

Basis: x is the node shifted left by Algorithm MC applied to SCH_0. Since x is shifted left, it follows that $s(x') < s(y')$. If x' and y' are in different underlying intervals, then the lemma clearly holds. If they are in the same underlying interval, then since no nodes have been shifted left prior to the shift of x, it follows from the definition of λ-feasible intervals and the fact that $s(x) = s(y)$ in SCH_0 that $s(x') = s(y')$, a contradiction.

Induction step: Assume the lemma holds for all previous shifts and let x be the node shifted left by Algorithm MC applied to SCH_{k-1}. Again, if x' and y' are in different underlying intervals, the lemma holds. So suppose that x' and y' are in the same underlying interval. Because $s(x) = s(y)$, it follows from the definition of λ-feasible intervals that x' had at least $s(y') - s(x')$ left shifts applied to it. Therefore, x' and y' had earlier been in the same conflicting set. By applying the induction assumption to x' and y', we get left (feas (x')) $\leqq$ left (feas (y')), which by the construction of λ-feasible intervals implies left (feas (x)) $\leqq$ left (feas (y)). □

LEMMA 7. *If Algorithm* MC (λ) *declares a problem instance infeasible in step* 4, *then there is no* λ*-feasible schedule for the problem instance.*

Proof. Assume for contradiction that Algorithm MC (λ) declares a feasible problem instance to be infeasible. Let S_t be the conflicting set on which MC (λ) halted, and let t be the time at which the nodes in S_t were scheduled. Define:

$$S'_{t,j} = \{x: t \leqq \text{left (feas } (x)) \text{ and right (feas } (x)) \leqq j\}.$$

Since the number of slots in the interval $[t, j]$ is $j - t + 1$, if $|S'_{t,j}| \geqq j - t + 2$ for some j, then by the pigeon hole principle, the problem instance is infeasible. So suppose that $|S'_{t,j}| \leqq j - t + 1$ for all values of j. Since this implies that $|S'_{t,t}| \leqq 1$, S_t contains at least two elements of $S'_{t,j}$ for some $j > t$.

Let J be the largest number greater than t such that Algorithm MC (λ) processes conflicting sets at times $J, J-1, \cdots, t+1, t$, and each of these conflicting sets contains an element of $S'_{t,J}$. It folllows from Lemma 6 that if node x is shifted left when a conflicting set is processed at time i, $t < i \leqq J$, and y is ultimately scheduled at time t, then left (feas (x)) $\leqq$ left (feas (y)). Therefore, by the definition of J, left (feas (y)) $\geqq t$. If right (feas (y)) $> J$, then the maximality of J is contradicted. Consequently, $t \leqq$ left (feas (y)) $\leqq$ right (feas (y)) $\leqq J$. Since at time t there are at least two nodes from $S'_{t,J}$, a simple counting argument gives $|S'_{t,J}| \geqq J - t + 2$, contradicting the assumption that $|S'_{t,j}| \leqq j - t + 1$ for all j. □

THEOREM 8. *If Algorithm* MC (λ) *halts in steps* 1 *or* 4, *then there is no* λ*-feasible schedule for the problem instance.*

Proof. The proof for step 1 follows from the correctness proof for SC, and the proof for step 4 follows from Lemma 7. □

The following theorem bounds the optimal value for λ.

THEOREM 9. *If there exists a feasible schedule for an instance of* MC, *with chains* $\mathscr{C} = \{C_1, \cdots, C_n\}$, *then there exists a* λ*-feasible schedule with* $\lambda_0 \leqq \lambda \leqq \lambda_0 + n - 1$ *where* $\lambda_0 = \max_{1 \leqq i \leqq n} \lambda(C_i)$, *and* $\lambda(C_i)$ *denotes the value of* λ *computed by Algorithm* SC *for chain* C_i.

Proof. The lower bound is obvious; the upper bound follows from Algorithm UPPER, presented below.

Algorithm UPPER.

If any interval has more nodes mapped into it than it has slots, declare the problem instance infeasible and halt.

Schedule C_1 according to Algorithm SC with $\lambda = \lambda_0$.

For $i = 2$ to n do

```
        Let {C_{i,1}, ···, C_{i,n_i}} be the nodes of C_i.
        Schedule C_{i,1} at the latest empty slot in f(C_{i,1})
        For j = 2 to n_i do
            Let t_0 = min (s(C_{i,j-1}) + λ_0 + i - 1, right (f(C_{i,j}))).
            If t_0 is an empty slot then set s(C_{i,j}) = t_0
            else do
            /* first test if nodes already scheduled can be shifted right */
              if there exists an empty slot at time t, t_0 < t ≦ right (f(C_{i,j})), do
                    let t_1 be the minimum such t
                    increase the start times of all nodes with start times t_0,
                    t_0 + 1, ···, t_1 by one /* shift right to create an empty slot */
                    Set s(C_{i,j}) = t_0
              od
              /* if right shift failed, then schedule node as late as possible
              within its feasible interval */
              else do
                    let t_2 be the largest valued empty slot with left (f(C_{i,j})) ≦
                    t_2 < t_0; if there is no such t_2, then declare failure and halt
                    Set s(C_{i,j}) = t_2
              od
            od
        od
    od
end
```

The proof of the correctness of UPPER follows from the succeeding lemmas. □

LEMMA 10. *In the ith iteration of the outer loop of* UPPER, $s^i(C_{k,j}) + i - 1 \geqq s(C_{k,j})$, $k \leqq i$, *where* $s^i(C_{k,j})$ *denotes the start time assigned to* $C_{k,j}$ *at the completion of the ith iteration of the outer loop of* UPPER *and* $s(C_{k,j})$ *denotes the start time assigned to* $C_{k,j}$ *by Algorithm* SC *with* $\lambda = \lambda_0$.

Proof. The proof is an induction on i. For $i = 1$ we have that all nodes $C_{1,j}$ $1 \leqq j \leqq n_1$ satisfy the lemma since $s^1(C_{1,j}) = s(C_{1,j})$. Assume that all nodes scheduled in the first $i-1$ iterations satisfy the lemma.

$C_{i,1}$ is scheduled so that $s^i(C_{i,1}) + i - 1 \geqq s(C_{i,1})$, because at most $i-1$ nodes can be scheduled to its right. Assume that all nodes up to and including the $C_{i,j}$ satisfy the inequality. In particular, $s^i(C_{i,j}) + i - 1 \geqq s(C_{i,j})$. This implies that $\min\{\text{right}(f(C_{i,j+1})),\ s^i(C_{i,j}) + i - 1 + \lambda_0\} \geqq \min\{\text{right}(f(C_{i,j+1})),\ s(C_{i,j}) + \lambda_0\}$. Note that the right hand side of the inequality is the expression for $s(C_{i,j+1})$, while the left hand side is the value of t_0 computed by UPPER. Hence, $t_0 \geqq s(C_{i,j+1})$.

There are three cases to consider. First, if UPPER schedules $C_{i,j+1}$ at t_0 with or without shifting any nodes, the lemma clearly holds for $C_{i,j+1}$. Second, if UPPER shifts some nodes right by one unit, the lemma still holds because by the induction hypothesis each such node satisfies $s^i(x) + i - 2 \geqq s(x)$ and shifting x by one unit to the right amounts to increasing $s^i(x)$ by one. Third, if UPPER schedules $C_{i,j+1}$ at some time $t_2 < t_0$, then t_2 cannot be more than $i-1$ units to the left of t_0, because at most $i-1$ nodes could have been scheduled contiguously. Therefore, $t_2 + i - 1 \geqq t_0$. Equivalently, $s^i(C_{i,j+1}) + i - 1 \geqq t_0 \geqq s(C_{i,j+1})$. This proves the lemma. □

THEOREM 11. *Algorithm* UPPER *constructs a feasible schedule if one exists.*

Proof. Assume a feasible schedule exists. By a pigeon hole argument, this implies that every interval has at least as many slots as it has nodes mapped into it. Therefore,

Algorithm UPPER will not determine infeasibility on the first test. Furthermore, each node $C_{i,j}$ will be assigned a start time so long as $s^i(C_{i,j-1})+\lambda_0+i-1 \geqq \text{left}(f(C_{i,j}))$.

We need only prove that no node falls short of the left endpoint of its interval. So suppose for contradiction that there exists a node $C_{i,j}$ such that $s^i(C_{i,j-1})+\lambda_0+i-1 < \text{left}(f(C_{i,j}))$. By the previous lemma, $s(C_{i,j-1}) \leqq s^i(C_{i,j-1})+i-1$. Hence, $s(C_{i,j-1})+\lambda_0 < \text{left}(f(C_{i,j}))$. This is a contradiction, because it implies that λ_0 is not sufficiently large to construct a feasible schedule for C_i using Algorithm SC. □

LEMMA 12. *There exists an instance of* MC *with chains* $C_1, \cdots, C_n$ *and intervals* I_1, I_2 *such that* $\lambda = \lambda_0 + n - 1$.

Proof. For each C_i there are two nodes $C_{i,1}$ and $C_{i,2}$ with $f(C_{i,1}) = I_1$ and $f(C_{i,2}) = I_2$. For this instance $\lambda_0 = \text{left}(I_2) - \text{right}(I_1)$. Further, for any ordering of the nodes in the first interval, the ordering of the nodes in the second interval that minimizes λ is the same as that of the first interval. This implies that $\lambda \geqq \lambda_0 + n - 1$, which in turn implies the lemma. □

THEOREM 13. *Given an instance of the* MC *problem, the running time required to determine the minimum value of* λ *for which a* λ*-feasible schedule exists is* $O(m(n \log n)^2)$.

Proof. We note that by Theorem 9 the optimum value of λ is bounded between λ_0 and $\lambda_0 + n - 1$. Therefore, the optimum value of λ can be determined in $O(\log n)$ iterations of Algorithm MC. The running time of Algorithm MC is dominated by the shifting of nodes in step 4. Since there are n chains, there can be no more than n nodes per interval. Therefore, the number of shifts per interval is bounded by $n(n-1)/2$. Since there can be no more than m intervals, the total number of shifts is $O(mn^2)$. Selecting the node to shift in step 4 can cost $O(\log n)$, thus giving a running time for a single iteration of algorithm MC of $O(mn^2 \log n)$. The bound follows from the first observation.[5] □

5. A general upper bound on the performance of a naive algorithm. Consider an instance of the Delay Problem with $B(N, A)$ being the DAG of precedence constraints. Partition the elements of $N(B)$ as follows.

N_s is the set of nodes in $N(B)$ that have only successors (sources);

N_p is the set of nodes in $N(B)$ that have only predecessors (sinks);

N_{ps} is the set of nodes in $N(B)$ that have both predecessors and successors;

N_0 is the set of nodes in $N(B)$ that have neither predecessors nor successors (singletons).

Let A be any naive algorithm for solving the Delay Problem with all of the following properties.

(1) First A schedules the nodes in N_s, with each node going in the rightmost empty slot in its interval;

(2) Then A schedules the nodes in N_p, with each node going in the leftmost empty slot in its interval;

(3) Finally, A schedules the nodes of N_{ps} and N_0 arbitrarily.

Let λ_A be the value that A computes for λ and let λ_{opt} be the optimal value of λ for the problem instance. Then we have the following theorem.

THEOREM 14. $\lambda_A \leqq 3\lambda_{\text{opt}} - 2$.

Proof. First observe that since each node in N_s has a successor, no node in N_s can be scheduled by A further than $\lambda_{\text{opt}} - 1$ from the right endpoint of its underlying interval, since otherwise some node in N_s would have its successor scheduled more

[5] Anderson, Munshi, and Simons [AMS] use preprocessing to obtain an algorithm for testing the existence of a λ-feasible schedule in time $O(n \log n)$. The result also holds for forests of either in-trees or out-trees.

than λ_{opt} away in any schedule. Similarly, any node in N_p cannot be scheduled by A further than $\lambda_{opt}-1$ from the left endpoint of its underlying interval.

Now, consider any arc $(y, z) \in A(B)$. Clearly, the length of this arc cannot be larger than right $(f(z))-$left $(f(y))$. Let S_{opt} be some optimal schedule, and define $s_{opt}(x)$ to be the start time of node x in S_{opt}.

Case 1. $y, z \in N_{ps}$. Let (x, y) and (z, w) be two neighboring arcs of (y, z). Clearly, right $(f(z)) \leqq$ left $(f(w))-1$ and right $(f(x)) \leqq$ left $(f(y))-1$. Adding these gives right $(f(z))-$left $(f(y)) \leqq$ left $(f(w))-$right $(f(w))-2$. Now, left $(f(w))-$right $(f(x)) \leqq s_{opt}(w)-s_{opt}(x) \leqq 3\lambda_{opt}$. Combining this with the previous inequality, we get right $(f(z))-$left $(f(y)) \leqq 3\lambda_{opt}-2$.

Case 2. $y \in N_s$ and $z \in N_{ps}$. Let (z, w) be a neighboring arc of (y, z). From the observation made at the beginning we know that right $(f(y))-s_A(y) \leqq \lambda_{opt}-1$, where $s_A(y)$ denotes the start time assigned to y by algorithm A. Since $s_{opt}(w)-s_{opt}(y) \leqq 2\lambda_{opt}$, we have that $s_{opt}(w)-s_A(y) \leqq 3\lambda_{opt}-1$, and consequently, left $(f(w))-s_A(y) \leqq 3\lambda_{opt}-1$. Also, since right $(f(z)) \leqq$ left $(f(w))-1$, we get right $(f(z))-s_A(y) \leqq 3\lambda_{opt}-2$. Hence, the maximum length of arc (y, z) is bounded by $3\lambda_{opt}-2$.

The remaining cases are similar. □

LEMMA 15. *Algorithm* A *runs in linear time.*

Proof. The partitioning into sets can clearly be done in linear time. Similarly, the actual scheduling can be done by always scheduling nodes at either the rightmost or leftmost empty slot in an interval. This scheduling requires only that two pointers be maintained for each interval, the updating of which can be performed in $O(1)$ time for each node scheduled. □

6. An approximation algorithm. We have implemented an approximation algorithm called APPROX. The basic idea of APPROX is to create a subgraph that we know how to schedule optimally, i.e., a set of chains. We iteratively refine the initial schedule for the subgraph to obtain a schedule for the entire graph B. During this process we attempt to prevent the value of λ from increasing by first trying reasonable shifts of the nodes, and increasing λ only when the shifts alone do not resolve conflicts.

The algorithm begins by pruning the graph as follows:

> If there exists p, q, r such that $(p, q), (q, r), (p, r) \in A(B)$, and $f(p)<f(q)<f(r)$, then the arcs (p, q) and (q, r) are eliminated.[6]

Subsequently, a greedy algorithm is used to create a set of spanning chains such that every node of $N(B)$ is contained in some chain. We then apply Algorithm MC to the set of all chains containing more than one node. Using the resulting schedule, we measure the length of each arc that is not an arc of some chain. If its length exceeds λ, we attempt to shift the nodes of one of the chains in a manner that preserves λ while still resulting in a feasible schedule. Failing this, we increase λ and continue. Finally, each "singleton" chain is inserted in a manner that attempts to minimize the lengths of the arcs incident to it.

7. Comparison with previous results. The *percentage parallelism* of a loop with n statements and delay λ is $(n-\lambda)/n$ [Cyt].

Figure 6 contains results obtained when our techniques were tested on some EISPACK routines, written in FORTRAN, that seem to exhibit very little parallelism. Listed is the percentage parallelism obtained after optimizing the code using the Illinois optimizing compiler but without using the DOACROSS feature that is currently a portion of that optimizing compiler (Unopt), Cytron's algorithm (CYT), and our algorithm (APPROX). Note that significant improvements over [Cyt] can be obtained

[6] Recall that we have reversed the direction of the backward loop carried arcs for the scheduling problem.

Name	Unopt	CYT	APPROX
CINVIT	6.3	6.3	9.6
COMLR2	11.1	11.1	11.1
COMQR2	5.6	5.6	11.1
HQR2	0	0	3.0
HTRIB3	4.0	4.0	33.3
HTRIDI	18.8	25.0	37.5
HTRID3	3.1	3.1	14.2
INVIT	0.0	2.9	5.8
MINFIT	8.0	8.0	20.0
QZVEC	23.1	36.3	36.3
SVD	0	0	14.3
TQLRAT	0	0	7.7

FIG. 6. *Percentage parallelism obtained on* EISPACK *routines.*

using our algorithm, and that in several cases where algorithm CYT was not able to find any parallelism in the code, our algorithm did find some parallelism. Indeed, in some cases our results are better by a factor greater than three. This does not actually contradict Theorem 14, since our technique for constructing independent sets tends to improve performance.

In summary, it has been shown that important special cases of the Delay Problem can be solved optimally in polynomial time. These solutions represent an improvement that is frequently considerably better than extant methods. Empirical evaluations show that in practice parallelism can be detected using our methods where none or little could be detected before.

Acknowledgments. We thank Ron Cytron for many helpful discussions and for a variant of Fig. 1, and Jeanne Ferrante and Nimrod Megiddo for valuable comments. We are also very grateful to Dave Alpern for his considerable assistance.

REFERENCES

[AK] R. ALLEN AND K. KENNEDY, *Automatic translation of* FORTRAN *programs to vector form*, TOPLAS, 9, (1987), pp. 491-542.

[AKPW] J. R. ALLEN, K. KENNEDY, C. PORTERFIELD, AND J. WARREN, *Conversion of control dependences to data dependences*, Proc. Symposium on Principles of Programming Languages, Austin, TX, 1983, pp. 177-189.

[All] J. R. ALLEN, *Dependence analysis for subscripted variables and its application to program transformations*, Ph.D. thesis, Computer Science Department, Rice University, Houston, TX, April, 1983.

[AMS] R. ANDERSON, A. MUNSHI, AND B. SIMONS, *A scheduling problem arising from loop parallelization on* MIMD *machines*, Third Aegean Workshop on Computing, Aegean Workshop on Computing 88, Corfu, Greece, June/July, 1988, pp. 124-133.

[Cyt] R. G. CYTRON, *Compile-time scheduling and optimization for asynchronous machines*, Ph.D. thesis, Computer Science Department, University of Illinois at Urbana-Champagne, Urbana, IL, 1984.

[KKLW] D. J. KUCK, R. H. KUHN, B. LEASURE, AND M. WOLFE, *The structure of an advanced vectorizer for pipeline processors*, IEEE Computer Society Fourth Internat. Computer Software and Applications Conference, 1980, pp 709-715

[Lam] L. LAMPORT, The Parallel Execution of DO Loops, Comm. ACM, 17 (1974), pp. 83-93.

[Mun] A. A. MUNSHI, unpublished manuscript.

[SAF] B. SIMONS, D. ALPERN, AND J. FERRANTE, *A foundation for sequentializing parallel code*, to appear.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 742–749, August 1990

013

ON FINDING AND VERIFYING LOCALLY OPTIMAL SOLUTIONS*

MARK W. KRENTEL†

Abstract. The problem of finding locally optimal solutions to combinatorial problems in the framework of PLS as defined by D.S. Johnson, C.H. Papadimitriou, and M. Yannakakis [*J. Comput. System Sci.*, 37(1988), pp. 79–100] is considered. A PLS-complete problem is exhibited such that the problem of *verifying* local optimality can be solved in LOGSPACE. For all previously known PLS-complete problems, verifying local optimality was P-complete, and it was conjectured in [*J. Comput. System Sci.*, 37(1988), pp. 79–100] that this was necessary.

Key words. computational complexity, local search, PLS, NP-complete

AMS(MOS) subject classifications. 68Q15, 68Q20, 68Q25

1. Introduction.

A recent approach to uncovering the structure of NP-complete problems [1] is the question of finding locally optimal solutions. Of course, if P $\neq$ NP then it is asking too much to find globally optimal solutions quickly, but even the question of local optimality is unclear.

To formalize this notion, Johnson, Papadimitriou, and Yannakakis [3] have defined a class of local search problems called PLS (Polynomial-Time Local Search). A PLS problem consists of a set of instances, feasible solutions, a cost function, and a neighborhood structure. A feasible solution is locally optimal if it has no neighbor with better cost. For example, consider GRAPH PARTITIONING with the 2-opt neighborhood. Instances are undirected graphs with an even number of vertices and weights on the edges. A feasible solution is a partition of the vertices into two equal-size pieces, and the cost function (which we are trying to minimize) is the sum of the weights on the edges that cross the partition. The 2-opt neighborhood consists of those partitions obtained by swapping one node from each side.

It is an open question if even locally optimal solutions for GRAPH PARTITIONING under 2-opt can be found in polynomial time. This problem is especially intriguing because it seems so natural that we should be able to find at least locally optimal solutions, yet it is not known. As a partial answer, we turn to completeness. Say that A is PLS-*reducible* to B if we can transform instances of A into instances of B such that given a local optimum for B, we can construct a local optimum for A. Reference [3] then shows that GRAPH PARTITIONING under the (much more complicated) Kernighan–Lin neighborhood structure [4] is PLS-complete. This result has two surprising corollaries. First, it is NP-hard to determine the output of the Kernighan–Lin algorithm on an arbitrary instance, and second, there are instances of GRAPH PARTITIONING for which Kernighan–Lin takes exponentially many iterations. Naturally, the completeness result also implies that local optima can be found in polynomial time under the Kernighan–Lin neighborhood structure only if local optima can be found for *all* problems in PLS. This result is especially important because, in practice, the Kernighan–Lin algorithm produces among the best feasible solutions of any heuristic algorithm [2].

The precise complexity of PLS problems is still open. Certainly they are no

* Received by the editors September 12, 1988; accepted for publication (in revised form) September 11, 1989. This research was supported in part by National Science Foundation grant CCR-8809370. An extended abstract was presented at the Fourth Annual IEEE Symposium on the Structure in Complexity Theory, Eugene, Oregon, June 1989.

† Department of Computer Science, Rice University, P.O. Box 1892, Houston, Texas 77251-1892.

harder than NP problems because globally optimal solutions can be found with an NP oracle. However, it is unlikely that they are NP-hard because if a PLS problem is NP-hard, then NP = coNP [3]. On the other hand, LINEAR PROGRAMMING with the simplex neighborhood is in PLS, and for simplex, local optimality implies global optimality. Although it would not imply any collapse, a polynomial-time algorithm for a PLS-complete problem would provide another proof that LINEAR PROGRAMMING is in P.

An essential feature of the construction in [3] is that, given a particular feasible solution, it is P-complete [6] to *verify* that the solution is locally optimal. The 2-opt neighborhood does not have this property; in fact, for 2-opt, local optimality can be verified in LOGSPACE. Also, since the 2-opt neighborhood is simpler than the Keringhan–Lin structure, it may be easier to find locally optimal solutions for 2-opt. Johnson, Papadimitriou, and Yannakakis conjectured that a problem could not be PLS-complete without the corresponding verification problem being P-complete. Actually, a lovely conjecture would be that local optima can be found in polynomial time if and only if the verification problem is in LOGSPACE, and in fact, this was our original motivation, to determine under what conditions local optima can be found in polynomial time.

In this paper, we disprove the above conjecture (under the assumption that P $\neq$ LOGSPACE) by exhibiting a PLS-complete problem such that the corresponding verification problem is in LOGSPACE. The problem we consider is CNF FLIP, i.e., given a Boolean formula in conjunctive normal form (CNF) with weights on the clauses, try to maximize the sum of the weights on the true clauses, under the neighborhood of flipping single variables. Reference [3] shows that the same problem using Boolean circuits (CIRCUIT FLIP), instead of formulas in CNF, is PLS-complete and that its verification problem is P-complete. Our result says that the P-completeness is not necessary and suggests that more problems are actually PLS-complete than was previously believed. We conjecture that GRAPH PARTITIONING under 2-opt is PLS-complete, but as of now, our result does not carry over. We leave it as an open problem under what conditions local optima can be found in polynomial time.

2. Definitions. The definitions below of PLS, and later PLS-reducible, are designed to formalize the question of when it is possible to find locally optimal solutions in polynomial time. Note that there are no constraints on the number of neighbors, or on the diameter of the neighborhood structure (the minimum distance between two feasible solutions, maximized over all pairs of feasible solutions), or even that the neighborhood relation is connected or symmetric. It would be valid to have every feasible solution a neighbor of every other feasible solution, but then algorithm C^{Π} would imply that testing global optimality is in polynomial time. Also note that C^{Π} is not required to compute the best neighbor.

DEFINITION. A PLS (Polynomial-Time Local Search) problem, Π, consists of the following.

(1) A set of *instances* $\mathcal{D}^{\Pi} \subseteq \Sigma^*$ for some finite alphabet Σ.
(2) A set of *feasible solutions* $\mathcal{FS}^{\Pi}(x) \subseteq \Sigma^{p(|x|)}$ for every $x \in \mathcal{D}^{\Pi}$, for some polynomial p.
(3) A set of *neighbors* $\mathcal{N}_x^{\Pi}(s) \subseteq \mathcal{FS}^{\Pi}(x)$ for every $x \in \mathcal{D}^{\Pi}$ and $s \in \mathcal{FS}^{\Pi}(x)$.
(4) A *measure* function $m_x^{\Pi} : \mathcal{FS}^{\Pi}(x) \to \mathbf{N}$ for every $x \in \mathcal{D}^{\Pi}$, where $\mathbf{N} = \{0, 1, 2, \cdots\}$, the set of natural numbers.

We require that $\mathcal{D}^{\Pi}$, $\mathcal{FS}^{\Pi}(x)$, and $\mathcal{N}_x^{\Pi}(s)$ be polynomial-time recognizable. There is no requirement that we can enumerate $\mathcal{N}_x^{\Pi}(s)$, and in fact, s may have more than

polynomially many neighbors. In addition, we require the following three algorithms to be computable in polynomial time.

(1) Algorithm A^Π, on input $x \in \mathcal{D}^\Pi$, produces an initial feasible solution in $\mathcal{FS}^\Pi(x)$.
(2) Algorithm B^Π, on input $x \in \mathcal{D}^\Pi$ and $s \in \mathcal{FS}^\Pi(x)$, computes $m_x^\Pi(s)$.
(3) Algorithm C^Π, on input $x \in \mathcal{D}^\Pi$ and $s \in \mathcal{FS}^\Pi(x)$, determines if s is locally optimal, and if not produces another solution $s' \in \mathcal{N}_x^\Pi(s)$ with better cost.

DEFINITION. CIRCUIT FLIP. Instances are Boolean circuits with n inputs and n outputs. A feasible solution is an assignment to the inputs, and the measure function (which we are trying to maximize) is the output sequence viewed as a binary number. The neighborhood of an assignment contains all other assignments obtained by flipping the value of a single input.

DEFINITION. CNF FLIP. Instances are Boolean formulas in conjunctive normal form with (binary) weights on the clauses. A feasible solution is an assignment to the variables, and the measure function (again, maximization) is the sum of the weights on the satisfied clauses. The neighborhood is again, all other assignments obtained by flipping the value of a single variable.

DEFINITION. A is PLS-*reducible* to B if there are polynomial-time computable functions f and g such that f maps instances of A to instances of B, and g given $x \in \mathcal{D}^A$ and a locally optimal $y \in \mathcal{FS}^B(f(x))$, produces a locally optimal $g(x, y) \in \mathcal{FS}^A(x)$. A problem is PLS-*complete* if it is in PLS and if all other problems in PLS are reducible to it.

Note that PLS reductions are transitive. Also, local optima can be found in polynomial time for a PLS-complete problem if and only if local optima can be found in polynomial time for *all* problems in PLS.

THEOREM 2.1. [3]. CIRCUIT FLIP *is* PLS-*complete, and its corresponding verification problem is* P-*complete.*

THEOREM 2.2. *Local optimality for* CNF FLIP *can be verified in* LOGSPACE.

Proof. Let $\Phi(x_1, \cdots, x_n)$ be a Boolean formula in CNF and let $A = (a_1, \cdots, a_n)$ be an assignment to $x_1, \cdots, x_n$. Then A is locally optimal for Φ if and only if for all $1 \leq i \leq n$, the sum of the weights on the clauses with x_i as the only satisfying literal is greater than or equal to the sum of the weights on the unsatisfied clauses containing x_i. It is straightforward to identify in LOGSPACE which clauses a variable appears in and which clauses we would gain or lose by flipping a variable. So it suffices to show how to add and compare n binary numbers each with n bits in LOGSPACE.

To add n numbers in LOGSPACE, first add the least significant bit of each of the numbers on the work tape. This uses only $O(\log n)$ space and gives the least significant bit of the result. Then, shift the result to the right by one bit and add the next least significant bit, and so on, keeping a "window" of $O(\log n)$ bits of the result. Of course, a LOGSPACE machine does not have the space to write down the entire answer, but it suffices to see the result bit by bit. □

3. Main result. This section gives the PLS-completeness construction for CNF FLIP, and in the next section, we draw corollaries for local search algorithms.

THEOREM 3.1. CNF FLIP *is* PLS-*complete.*

Proof. The problem is clearly in PLS, so it suffices to reduce CIRCUIT FLIP to CNF FLIP. Let C be a circuit with inputs $X_1, \cdots, X_n$, outputs $Y_1, \cdots, Y_n$, and gates $G_1, \cdots, G_M$. Assume for convenience that $M \geq n$ and that the first n gates, $G_1, \cdots, G_n$, just copy the input and that these are the only gates in which $X_1, \cdots, X_n$

appear. We also assume that the gates are topologically ordered, so that if G_i is an input to G_j, then $i < j$.

Variables. We reduce C to a CNF Boolean formula Φ with variables $x_1, \cdots, x_n$, $f_1, \cdots, f_n$, $g_1, \cdots, g_M$, and $t_1^i, \cdots, t_M^i$ for each $1 \le i \le n$. Obviously, the variables $x_1, \cdots, x_n$ represent the input to the circuit, and $g_1, \cdots, g_M$ represent the output of the gates. For each input x_i, the variables $t_1^i, \cdots, t_M^i$ (called test circuits) represent the new output of C's gates if we were to flip the value of x_i. Since $G_1, \cdots, G_n$ just copy the input, in general we want $g_j = t_j^i = x_j$ for $1 \le j \le n$, except that we want $t_i^i = \overline{x}_i$. The variables $f_1, \cdots, f_n$ are used to flip the inputs $x_1, \cdots, x_n$. Normally, all f_i will be zero, but when we are ready to flip x_i, we first set f_i to one and then flip x_i and reset $g_1, \cdots, g_M$, and then reset f_i to zero. This allows us to switch to the circuit's new output from $t_1^i, \cdots, t_M^i$ with a *single* flip of f_i, and will be a key point in the construction.

Clauses. The clauses are divided into three classes: *hard constraints*, *medium weights*, and *small weights*. The hard constraints have the heaviest weight, and it will turn out that if any hard constraint is violated, then it will always be possible to improve the weight by flipping a single variable. Thus, a feasible solution *must* satisfy the hard constraints in order to have any hope of being locally optimal. The medium weights represent the output of the circuit, and the small weights give a credit for setting the test circuits correctly.

Hard constraints. The largest constraint is that at most one f_i can be set to one. This can be expressed as

$$\bigwedge_{1 \le i < j \le n} (\overline{f}_i + \overline{f}_j)^{2^{7M}}, \tag{1}$$

where $(\psi)^w$ means that all clauses in ψ have weight w.

The second hard constraint is that the output of the circuit is computed correctly. If all $f_i = 0$, then we use $g_1, \cdots, g_M$ to compute the circuit, and if some $f_i = 1$ then we use $t_1^i, \cdots, t_M^i$. Thus, the constraint we wish to express is

$$(\text{gate } 1 \text{ correct})^{2^{6M}} \cdots (\text{gate } M \text{ correct})^{2^{5M+1}}, \tag{2}$$

where "gate j correct" means

$$\left(\left(\bigvee_{1 \le i \le n} f_i = 0 \right) \Rightarrow (g_j \text{ correct}) \right) \wedge \left(\bigwedge_{1 \le i \le n,\, i \ne j} \left((f_i = 1) \Rightarrow (t_j^i \text{ correct}) \right) \right). \tag{3}$$

For example, consider the gate "$g_j = g_a \wedge g_b$." This can be expressed in CNF as

$$(g_j + \overline{g}_a + \overline{g}_b)(\overline{g}_j + \overline{g}_a + g_b)(\overline{g}_j + g_a). \tag{4}$$

So, the constraint "gate j correct" can be expressed as

$$(f_1 + \cdots + f_n + g_j + \overline{g}_a + \overline{g}_b)(f_1 + \cdots + f_n + \overline{g}_j + \overline{g}_a + g_b)(f_1 + \cdots + f_n + \overline{g}_j + g_a)$$

$$\wedge \left(\bigwedge_{1 \le i \le n,\, i \ne j} (\overline{f}_i + t_j^i + \overline{t}_a^i + \overline{t}_b^i)(\overline{f}_i + \overline{t}_j^i + \overline{t}_a^i + t_b^i)(\overline{f}_i + \overline{t}_j^i + t_a^i) \right). \tag{5}$$

A similar construction is possible for the "or" and "not" gates. Note that there is no constraint that $t_i^i = \overline{x}_i$. This will be necessary for allowing x_i to be flipped at the appropriate time.

Medium weights. The next largest constraints, the medium weights, represent the outputs $Y_1, \cdots, Y_n$. Recall that the output of the circuit is represented in a subset of the gates and that the output is taken from $g_1, \cdots, g_M$ if $\bigvee f_i = 0$, or from $t_1^i, \cdots, t_M^i$ if $f_i = 1$. These weights are computed similarly to the constraints that the gates are set correctly and can be expressed as

$$(\text{output } 1 \text{ is } 1)^{2^{4M+n}} \cdots (\text{output } n \text{ is } 1)^{2^{4M+1}}. \tag{6}$$

For example, if g_j is an output gate, "output j is 1" is expressed as

$$(f_1 + \cdots + f_n + g_j) \wedge \left(\bigwedge_{1 \le i \le n} (\overline{f}_i + t_j^i) \right). \tag{7}$$

Small weights. There are three groups of small weights. The largest gives a credit for setting $x_i = t_i^i$ when $f_i = 1$ and can be expressed as

$$\bigwedge_{1 \le i \le n} (\overline{f}_i + \overline{x}_i + t_i^i)^{2^{3M}} (\overline{f}_i + x_i + \overline{t}_i^i)^{2^{3M}}. \tag{8}$$

The second group gives a credit for setting $f_i = 0$ and can be expressed as

$$(\overline{f}_1)^{2^{2M}} \cdots (\overline{f}_n)^{2^{2M}}. \tag{9}$$

The third and smallest group gives a credit for setting the values of the test circuits correctly, except that here, "t_i^i correct" means $t_i^i = \overline{x}_i$. This can be expressed as

$$(g_1 \text{ correct})^{2^M} \cdots (g_M \text{ correct})^{2^1} \wedge \left(\bigwedge_{1 \le i \le n} (t_1^i \text{ correct})^{2^M} \cdots (t_M^i \text{ correct})^{2^1} \right), \tag{10}$$

where "g_j correct" is expressed as in (4). Then Φ is the product of the clauses in (1), (2), (6), (8), (9), and (10).

Remark. Note that the weights are constructed in descending powers of two. This technique makes it easier to argue about the structure of a local optimum because a higher power of two will completely dominate all of the lower powers. Another point is that the CNF constraints in (1), (3), (4), (5), (7), (8), and (9) are carefully constructed so that if a constraint is violated, then exactly one of the clauses is unsatisfied. Although it may take several clauses to express one constraint, the *difference* between satisfying the constraint and violating it is just the weight on a single clause. Thus, we may pretend that each constraint is expressed by a single clause.

Claim. Let B be a locally optimal assignment for Φ. Then B corresponds to a locally optimal solution for C.

First we claim that at most one $f_i = 1$ in B. If not, then the weight from (1) can be improved by a *single* flip of some f_i to zero. Second, we claim that the gates representing the circuit, either $g_1, \cdots, g_M$ or $t_1^i, \cdots, t_M^i$, are correct. Again, if not, the weight from (2) can be improved by flipping the lowest numbered incorrect gate; and again, this weight dominates any other weight that could be lost. Therefore, B

must satisfy the hard constraints. This implies that B corresponds to *some* feasible solution A for C, and also that the output of A can be determined from the weight of B. (Although there is no constraint that $t_i^i = \overline{x}_i$ if $f_i = 1$, use t_i^i for the value of input i in A. Later we will see that all f_i must be zero in B anyway.)

The remainder of the proof splits depending on whether or not some $f_i = 1$. Suppose first that all $f_i = 0$; later we will see that the second case is actually not possible. Then we already know that $g_1, \cdots, g_M$ are correct. Suppose that some t_j^i is incorrect (including the case that t_i^i is incorrect, meaning $t_i^i \neq \overline{x}_i$). Because all $f_i = 0$, the test gates play no part in the hard constraints, the medium weights, or in the largest of the small weights. Thus, flipping t_j^i improves the weight of B from (10) and it does not destroy any of the heavier weights. So, the test gates must be correct. This says that the new output of flipping any one of the inputs is available in B by the *single* flip of some $f_i = 1$. Because the test gates are correct, flipping $f_i = 1$ would not violate the hard constraints, but it would represent the new medium weights. Thus, if B is locally optimal, then flipping one input cannot improve the output in C, and thus A is locally optimal for C.

Now suppose that B is locally optimal and that $f_i = 1$ for some $1 \leq i \leq n$. From the hard constraints, we already know that $t_1^i, \cdots, t_M^i$ are correct and reflect the circuit's output. Here it is $g_1, \cdots, g_M$ that play no role in the hard constraints or the medium weights. First, $f_i = 1$ implies that x_i does not affect the hard constraint for t_i^i correct (this was the reason for omitting the case $i = j$ in (5)). So, if $x_i \neq t_i^i$, then we can improve B's weight by flipping $x_i = t_i^i$, because the weight of (8) dominates the weight of any other clause containing x_i. Therefore, $x_i = t_i^i$ in B. Then, if any g_i is incorrect, we can improve B's weight by flipping the incorrect gate because $g_1, \cdots, g_M$ does not affect the hard constraints as long as $f_i = 1$. Thus, $g_1, \cdots, g_M$ must be correct, and then $x_i = t_i^i$ implies that $g_1, \cdots, g_M$ must have the identical values as $t_1^i, \cdots, t_M^i$. But then, flipping f_i to zero preserves the hard constraints and the medium weights and also picks up the weight from (9). This implies that B is not locally optimal; and, in fact, all $f_i = 0$ in any locally optimal solution for Φ. This completes the proof. □

4. Local search algorithms. This result has implications for local search algorithms for CNF FLIP; in fact, these corollaries were one of the original motivations for defining PLS. Associated with every PLS problem, Π, there is a "standard" local search algorithm. Given an instance $x \in \mathcal{D}^\Pi$, start at the solution produced by $A^\Pi(x)$ and continue to move to the solution produced by C^Π until locally optimal. Since the set of feasible solutions is finite, this algorithm always terminates, and by hypothesis it takes polynomial time per iteration. So the question is how many iterations will be necessary.

There are PLS problems and instances for which the standard algorithm takes exponentially many iterations, and in fact, for which computing the output of the standard algorithm is an NP-hard function. Johnson, Papadimitriou, and Yannakakis [3] give the following example (although without naming it).

DEFINITION. LINEAR SAT. Instances are Boolean formulas in conjunctive normal form, and feasible solutions are assignments to the variables considered either as strings over $\{0, 1\}$ or as binary numbers. The cost of solution s (to be minimized) is 0 if s is a satisfying assignment and $s + 1$ otherwise. The neighborhood of s is $\{s - 1\}$ for $s \neq 00 \cdots 0$ and $\emptyset$ for $s = 00 \cdots 0$. The initial feasible solution is $11 \cdots 1$.

Due to the degenerate neighborhood structure, the standard algorithm for LINEAR SAT is forced to exhaustively search through all feasible solutions. Furthermore, the

output of the standard algorithm starting at $11\cdots1$ is either solution $00\cdots0$ with cost 1 if the formula is not satisfiable, or some other solution with cost 0 if the formula is satisfiable. Thus computing the output of the standard algorithm is an NP-hard function, and there are instances for which the standard algorithm takes exponentially many iterations. It is interesting to note that, although it is hard to compute the local optimum produced by the standard algorithm, this does not imply that it is necessarily hard to find some local optimum. In fact, $00\cdots0$ is always locally optimal for LINEAR SAT.

These properties for local search algorithms generally carry over to PLS-complete problems. For CIRCUIT FLIP, it is NP-hard to compute the output of the standard algorithm and there are examples that take exponential time [3]. This result does not follow directly from the definition of PLS-reduction; it is necessary to analyze the reduction from LINEAR SAT to CIRCUIT FLIP. Essentially, the neighborhoods in CIRCUIT FLIP embed the neighborhoods in LINEAR SAT, with a path in CIRCUIT FLIP corresponding to a path in LINEAR SAT. This forces a local search algorithm for CIRCUIT FLIP to simulate the standard algorithm for LINEAR SAT. Furthermore, these results hold for any local search algorithm using *any* local improvement rule, even an omniscient one. As long as the algorithm obeys the neighborhood structure (i.e., only moves to a neighbor of the current solution with improved cost), a path in CIRCUIT FLIP will correspond to a path in LINEAR SAT.

THEOREM 4.1. (i) *It is* NP*-hard to compute the output of the standard algorithm for* CNF FLIP *on arbitrary instances and starting points, and* (ii) *there are instances and starting points for* CNF FLIP *such that* any *local search algorithm takes exponentially many iterations before reaching a local optimum.*

Proof. These results are proved in [3] for the CIRCUIT FLIP problem. It suffices to show that the reduction to CNF FLIP embeds the same neighborhood structure as CIRCUIT FLIP and that a local search path for CNF FLIP corresponds to a path in CIRCUIT FLIP. Say that an assignment is *stable* if it satisfies all of the hard constraints and all $f_i = 0$. A stable assignment corresponds to an assignment for CIRCUIT FLIP. Starting from a stable assignment, as long as all $f_i = 0$, no local search algorithm can flip any x_j because this would violate the constraint that g_j is correct. So, in order to make progress, the algorithm must at some point flip some f_i to one. Then, as long as $f_i = 1$, no algorithm can flip any x_j for $i \neq j$ because this would violate the constraint that t_j^i is correct. So, at some later point, the algorithm must flip f_i back to zero. Thus, the algorithm has simulated a flip in CIRCUIT FLIP, and so a local search path for CNF FLIP corresponds to a path for CIRCUIT FLIP. □

5. An update. After submitting this paper, the author has learned of substantial progress on PLS and several new complete problems. Schäffer and Yannakakis [9] have extended the result for CNF FLIP to formulas with bounded size clauses and also to POSITIVE NOT-ALL-EQUAL 3-SAT (a special case of 2-CNF FLIP). As corollaries they can show PLS-completeness for GRAPH PARTITIONING under 2-opt, MAX CUT, and also STABLE CONFIGURATION FOR CONNECTIONIST NETWORK. Inspired by these results, the author extended the result for CNF FLIP to formulas with simultaneously bounded size clauses and bounded number of occurrences of variables [5]. A corollary is that TRAVELING SALESMAN is PLS-complete under the k-opt neighborhood for sufficiently large k. Also, Papadimitriou [7] has a construction for TRAVELING SALESMAN under the Lin–Kernighan algorithm based on 3-opt moves. The results from [7] and [9] appear together in [8].

REFERENCES

[1] M. GAREY AND D. S. JOHNSON, *Computers and Intractability: A Guide to the Theory of* NP-*Completeness*, W. H. Freeman, San Francisco, CA, 1979.

[2] D. S. JOHNSON, C. R. ARAGON, L. A. MCGEOCH, AND C. SCHEVON, *Optimization by simulated annealing: An experimental evaluation* (*part* I), manuscript, 1987.

[3] D. S. JOHNSON, C. H. PAPADIMITRIOU, AND M. YANNAKAKIS, *How easy is local search?*, J. Comput. System Sci., 37 (1988), pp. 79–100.

[4] B. W. KERNIGHAN AND S. LIN, *An efficient heuristic procedure for partitioning graphs*, Bell Systems Tech. J., 49 (1972), pp. 291–307.

[5] M. W. KRENTEL, *Structure in locally optimal solutions*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, Research Triangle Park, NC, 1989, pp. 216–221.

[6] R. LADNER, *The circuit value problem is* log *space complete for* P, SIGACT News, 7 (1975), pp. 18–20.

[7] C. H. PAPADIMITRIOU, *The complexity of the Lin–Kernighan heuristic for the traveling salesman problem*, 1989, submitted for publication.

[8] C. H. PAPADIMITRIOU, A. A. SCHÄFFER, AND M. YANNAKAKIS, *On the complexity of local optimality*, in Proc. 22nd Annual ACM Symposium on Theory of Computing, Baltimore, MD, 1990.

[9] A. A. SCHÄFFER AND M. YANNAKAKIS, *Simple local search problems that are hard to solve*, 1989, submitted for publication.

SIAM J. COMPUT.
Vol. 19, No. 4, pp. 750-773, August 1990

014

THE STRUCTURE OF POLYNOMIAL IDEALS AND GRÖBNER BASES*

THOMAS W. DUBÉ†

Abstract. This paper introduces the cone decomposition of a polynomial ideal. It is shown that every ideal has a cone decomposition of a standard form. Using only this and combinatorial methods, the following sharpened bound for the degree of polynomials in a Gröbner basis can be produced. Let $K[x_1, \cdots, x_n]$ be a ring of multivariate polynomials with coefficients in a field K, and let F be a subset of this ring such that d is the maximum total degree of any polynomial in F. Then for any admissible ordering, the total degree of polynomials in a Gröbner basis for the ideal generated by F is bounded by $2((d^2/2) + d)^{2^{n-1}}$.

Key words. Gröbner bases, algebraic computation, Hilbert functions

AMS(MOS) subject classifications. 68Q40, 05A17

1. Introduction.

Many problems of symbolic computation can ultimately be reduced to determining if a given polynomial p is contained in the ideal generated by a set of polynomials F. Gröbner bases are special bases for polynomial ideals with several important computational properties including the ability to rapidly determine ideal membership. The term *Gröbner basis* was coined by Buchberger, who earlier had pioneered the idea in his thesis. Gröbner bases differ only slightly from the *standard bases* defined by Hironaka, and many of these concepts can be traced back to the H-bases of Macaulay.

The increasing interest in Gröbner bases as a computational tool is in large part due to the algorithm provided by Buchberger whereby for any set of polynomials F, it is possible to construct a Gröbner basis for the ideal generated by F.

Although modified versions of Buchberger's algorithm have shown success in practice (including some commercial systems), the complexity of the algorithm has not been well understood. Giusti [6] has shown a Gröbner basis construction that always produces a Gröbner basis containing only polynomials of the lowest possible degree. A first step in understanding the complexity of the algorithm then is to bound the degree of polynomials that occur in a minimal Gröbner basis.

It has been widely known (thanks to [8] and [10]) that in the worst case the degree of polynomials in a Gröbner basis is at least double exponential in the number of indeterminates in the polynomial ring. This lower bound precludes the existence of an upper bound that would show the Gröbner basis algorithm to be tractable, but it does not answer the following question: "How large can the polynomials in a Gröbner basis be?"

The direction for producing an upper bound was provided by Bayer [1]. Bayer's thesis, together with the results of [6] and [10], shows that the degree bound of elements in a Gröbner basis is bounded by $(2d)^{(2n+2)^{n+1}}$.

The steps in producing this former bound may be summarized as follows:

(1) Begin with a basis F for $I \subseteq K[x_1, \cdots, x_n]$, with d the maximum degree of polynomials in F.

(2) If the ideal I is affine, introduce a new variable x_{n+1} to homogenize the ideal I to ${}^hI$.

* Received by the editors June 27, 1988; accepted for publication October 5, 1989. This work was supported in part by National Science Foundation grants DCR-84-01898 and DCR-84-01633.

† Courant Institute, New York University, New York, New York 10012.

(3) Place ${}^hI$ into *generic coordinates* [1]. Here it must be assumed that K is of characteristic zero.

(4) In generic coordinates the degree of polynomials required in a Gröbner basis with respect to reverse lexicographic ordering is bounded by $(2d)^{2^{n-1}}$ ([6]).

(5) The degree bound in generic coordinates also serves as a bound on the *regularity* of ${}^hI$ ([6]). (An ideal has regularity m if for every degree m polynomial p, the ideal (I, p) has a different Hilbert polynomial than I.) Since ${}^hI$ has $n+1$ variables, the regularity m of ${}^hI$ is bounded by $(2d)^{2^n}$.

(6) A polynomial ideal over n variables with regularity m has its Macaulay constant (b_1 as used in this paper) bounded by $(m + 2n + 2)^{(2n+2)^n}$ ([10]). The Macaulay constant of ${}^hI$ is therefore bounded by $D = ((2d)^{2^n} + 2n + 2)^{(2n+2)^n} \approx (2d)^{(2n+2)^{n+1}}$.

(7) For any admissible ordering, the degree of polynomials in a Gröbner basis for ${}^hI$ is bounded by the maximum of m and b_1 ([1]). The degree of these polynomials is therefore bounded by the value D given above.

(8) Specializing ${}^hI$ back to I by setting $x_{n+1} = 1$ produces a Gröbner for I whose polynomials also satisfy this same degree bound.

A first remark concerning this procedure is that bounding the regularity of ${}^hI$ is an unnecessary detour. Reference [6] shows that in generic coordinates with respect to reverse lexicographic ordering a Gröbner basis G can contain a polynomial g with $\mathrm{Hterm}(g) \in \mathrm{PP}[x_1, \cdots, x_i]$ only if for every degree z such that $d \leq z \leq \deg(g)$, G contains a polynomial g_d with $\deg(g_d) = d$ and $\mathrm{Hterm}(g_d) \in \mathrm{PP}[x_1, \cdots, x_i]$. This condition is nearly equivalent to what is defined in this thesis as a *standard cone decomposition*, and in fact the standard cone decomposition was developed as a way to mimic this behavior. Directly from Giusti's decomposition, the Hilbert polynomials of $\tilde{I}$ and $K[X]/\tilde{I}$ can be written in the forms needed to produce the bound on the Macaulay constant given in Chapter 3.

Furthermore, the methods for obtaining the old bound use fairly specialized branches of commutative algebra and algebraic geometry. Expertise in these areas is not common among computer scientists. Since there are a growing number of computer scientists who will want to use Gröbner bases, there is a need for a self-contained treatment. The methods of algebraic geometry are concise and elegant, but there is often much more insight gained by using brute force.

A major result of this current study was to obtain a new upper bound for Gröbner basis degree. If F is a set of n variable polynomials of degree at most d, then we prove that a reduced Gröbner basis for the ideal generated by F has degree at most

$$2\left(\frac{d^2}{2} + d\right)^{2^{n-1}}.$$

I believe that the method of obtaining this bound is perhaps of greater importance than the bound itself. The method, which involves decomposing the ideal into disjoint *cones*, avoids the need to change to generic coordinates. This greatly simplifies the description of the proof and eliminates the requirement that the field K have characteristic zero. Moreover, it sheds much light onto the structure of an ideal I and the quotient ring $K[X]/I$, and it is expected that further applications for cone decompositions could be found.

2. Background. The material in this section is presented primarily for the purpose of establishing notations and terminology. For a more thorough introduction to

Gröbner basis, the reader is directed to [2], [3], [4], or [9]. The notations for homogeneity and Hilbert functions are borrowed primarily from [12] and [11], and a more detailed summary of all this information can be found in [5].

2.1. Admissible orderings and Gröbner bases.

DEFINITION. A total ordering $\underset{A}{\geq}$ on the power products $\mathrm{PP}[X] = \mathrm{PP}[x_1, \cdots, x_n]$ of the ring $\mathcal{A}$ is called an *admissible ordering* if the following axioms hold:

(1) For all power products $a, b, c \in \mathrm{PP}[X]$, $a \underset{A}{>} b \Longrightarrow ca \underset{A}{>} cb$.

(2) For all variables x_i, $x_i \underset{A}{>} 1$.

Closely related to the concept of admissible orderings is that of head terms. The $\underset{A}{>}$-greatest power product contained in a monomial of a polynomial h is called the *head term of h with respect to* $\underset{A}{>}$ and is denoted by $\mathrm{Hterm}_A(h)$. For an ideal I, $\mathrm{Head}_A(I)$ is used to denote the ideal generated by the set $\{\mathrm{Hterm}_A(h) : h \in I\}$.

DEFINITION. Let G be a basis for the ideal I and let $\underset{A}{>}$ be an admissible ordering. G is called a *Gröbner basis* of I (with respect to $\underset{A}{>}$) if $\mathrm{Head}_A(I)$ is generated by the set $\{\mathrm{Hterm}(g) : g \in G\}$.

Let F be a set of polynomials and $\underset{A}{>}$ a fixed admissible ordering. A polynomial h is said to be *F-reducible*, if there exists $f \in F$, and monomial $c \in \mathcal{A}$ such that $\mathrm{Hterm}_A(cf)$ is a monomial of h. The polynomial $g = h - cf$ is then called a reduct of h, and this relationship is denoted as $h \xrightarrow{F} g$. The transitive closure $h \xrightarrow[*]{F} g$ of the reduction operation is defined to mean that there exists a sequence of polynomials $p_1, \cdots, p_k$ such that $p_1 = h$, $p_k = g$, and for all $i < k$, $p_i \xrightarrow{F} p_{i+1}$. Finally, g is called an *F-normal form of h* if

(1) $h \xrightarrow[*]{F} g$, and

(2) g is not F-reducible.

The following conditions are all equivalent (e.g., [3], [9]):

(1) G is a Gröbner basis for I with respect to $\underset{A}{>}$.

(2) $G \subset I$ and for every $h \in I$ there exists a $g \in G$ such that $\mathrm{Hterm}_A(g)$ divides $\mathrm{Hterm}_A(h)$.

(3) For all $h \in \mathcal{A}$, 0 is a G-normal form of h if and only if $h \in I$.

(4) G is a basis for I and every $h \in \mathcal{A}$ has a unique G-normal form that may be denoted as $\mathrm{nf}_G(h)$.

One of the most important features of Gröbner bases is the existence of unique normal forms. The following lemma shows that these normal forms provide a system of representatives for the residue class ring $\mathcal{A}/I$.

LEMMA 2.1. *Let G be a Gröbner basis for I with respect to the admissible ordering $\underset{A}{>}$. Then the following properties hold for all $s, t \in \mathcal{A}$:*

(1) $s - \mathrm{nf}_G(s) \in I$.

(2) $s - t \in I \iff \mathrm{nf}_G(s) = \mathrm{nf}_G(t)$.

(3) $\mathrm{nf}_G(s + t) = \mathrm{nf}_G(s) + \mathrm{nf}_G(t)$.

In a slight abuse of notation, N_I will be used to denote the set of normal forms

$$N_I = \{\mathrm{nf}_G(a) : a \in \mathcal{A}\},$$

where G is an arbitrary fixed Gröbner basis for I.

2.2. Direct decompositions. Let T be a subset of the polynomial ring $\mathcal{A}$, and let $S_1, \cdots, S_m$ be a (possibly infinite) family of subsets of T. The sets S_i are said to be a *direct decomposition* of T if every $p \in T$ can be *uniquely* expressed in the form $p = \sum_{i=1}^{r} p_i$, where $p_i \in S_i$ and r is finite. The fact that the S_i form a direct decomposition of T is expressed using the notation

$$T = S_1 \oplus S_2 \oplus \cdots \oplus S_m .$$

The following two important properties of direct decompositions can be easily verified.

(1) Let $S_1, \cdots, S_k$ be a direct decomposition for T, and let $R_1, \cdots, R_m$ be a direct decomposition for S_1. Then $S_2, \cdots, S_k, R_1, \cdots, R_m$ is a direct decomposition for T.

(2) Let $P = \{hf : h \in T\}$ for some polynomial f, and let $S_1, \cdots, S_k$ be a direct decomposition of T. Then the sets $Q_i = \{hf : h \in S_i\}$ form a direct decomposition of P.

Example 1. For any ideal $I \subseteq \mathcal{A}$, I and N_I form a direct decomposition of $\mathcal{A}$.

Proof. Let G be the Gröbner basis of I used to form $N_I = \mathrm{nf}_G(\mathcal{A})$. Since G is a Gröbner basis, each polynomial h has a unique G-normal form, and the decomposition $h = \mathrm{nf}_G(h) + (h - \mathrm{nf}_G(h))$ is unique. □

DEFINITION. Let I be any ideal of $\mathcal{A}$, and $h \in \mathcal{A}$. The ideal quotient operation $I : h$ is defined by $I : h = \{f \in \mathcal{A} : fh \in I\}$. Note that it trivially follows that $(I : g) : h = I : (gh)$.

Example 2. For an ideal $J \subset \mathcal{A}$ and $f \in \mathcal{A}$, let

$$\begin{aligned} I &= (J, f), \quad L = J : f, \\ S &= \{af : a \in N_L\} ; \end{aligned}$$

then $I = J \oplus S$.

Proof. Let G be the Gröbner basis for L used to form N_L and $S = fN_L$. The sets J and S are clearly subsets of I, so it need only be shown that each $h \in I$ can be uniquely expressed as $h = h_J + h_S$. It will first be shown that such a decomposition exists, and then that the decomposition is unique.

Every polynomial $h \in I$ can be written as $h = a_J + a_f f$ with $a_J \in J$. It is now claimed that a decomposition of h exists with $h_J = h - \mathrm{nf}_G(a_f)f$ and $h_S = \mathrm{nf}_G(a_f)f$. Since the sum of these two polynomials is trivially h, it must only be shown that $h_J \in J$. This follows directly from the definitions of the sets involved:

$$\begin{aligned} &a_f - \mathrm{nf}_G(a_f) \in L, \\ &(a_f - \mathrm{nf}_G(a_f))f \in J, \\ &h_J = a_J + (a_f - \mathrm{nf}_G(a_f))f \in J . \end{aligned}$$

Now consider any two decompositions of h:

$$h = a_1 + \mathrm{nf}_G(b_1)f_r = a_2 + \mathrm{nf}_G(b_2)f_r ,$$

where $a_1, a_2 \in J$.

$$\begin{aligned} &(\mathrm{nf}_G(b_1) - \mathrm{nf}_G(b_2))f_r = a_2 - a_1 \in J, \\ &\mathrm{nf}_G(b_1) - \mathrm{nf}_G(b_2) \in L, \\ &\mathrm{nf}_G(b_1) - \mathrm{nf}_G(b_2) = 0 . \end{aligned}$$

Therefore the decomposition is unique. □

Applying this technique recursively, we obtain the following decomposition of an ideal.

Example 3. Let $F = \{f_1, \cdots, f_r\}$ be a basis for an ideal I. Let S_1 be the principal ideal (f_1), and for each $i = 2, \cdots, r$, let

$$\begin{aligned} L_i &= (f_1, \cdots, f_{i-1}) : f_i, \text{ and} \\ S_i &= \{hf_i : h \in N_{L_i}\} . \end{aligned}$$

Then, $I = S_1 \oplus \cdots \oplus S_r$.

In summary, for any ideal I, the ring $\mathcal{A}$ can be decomposed into I and N_I. Furthermore, I itself can be decomposed into sets of the form $S_i = \{hf_i : h \in N_{L_i}\}$, which in turn could be further decomposed if we could decompose N_{L_i}. Sets of the form N_I need to be studied more closely.

2.3. Homogeneity. Let f be a polynomial in $\mathcal{A}$; then f can be written as a finite sum $f = f_k + f_{k-1} + \cdots + f_0$, where each f_z is either zero, or a sum of monomials each of which has total degree z. In such a decomposition of f, each nonzero f_z is called the *homogeneous component of f of degree z*. The nonzero homogeneous component f_k of greatest total degree is called the initial form of f and is denoted by $\text{in}(f)$. A polynomial f is called a *homogeneous polynomial* if f consists of at most one nonzero homogeneous component.

DEFINITION. A set $S \subseteq \mathcal{A}$ is called *homogeneous* if it satisfies the following two properties:

(1) $f \in S$ implies that each homogeneous component of f is also in S.

(2) f is a K-module.

A homogeneous set S that is an ideal of $\mathcal{A}$ is called simply a *homogeneous ideal.* A direct decomposition $S_1, \cdots, S_r$ of a homogeneous set T is called a *homogeneous direct decomposition* if each S_i is homogeneous.

For a homogeneous set T, the subset of degree z homogeneous polynomials will be denoted by T_z, i.e.,

$$T_z = \{f \in T : f \text{ is homogeneous of degree } z\} .$$

If T is closed under addition, then the collection of sets $\{T_0, T_1, \cdots\}$ trivially form a homogeneous direct decomposition of T.

For p a polynomial in the affine ring $\mathcal{A} = K[x_1, \cdots, x_n]$, let p be written as a sum of monomials $p = p_1 + \cdots + p_m$. The homogenization function ${}^h p$ is a mapping from the affine ring $\mathcal{A}$ to the projective ring $K[x_1, \cdots, x_n; y]$ where y is a new variable and the mapping is defined as

$${}^h p = \sum_{i=1}^{m} p_i y^{\deg(p) - \deg(p_i)}.$$

Throughout this paper, y will be used to denote the extra variable, which is introduced by homogenization. ${}^h\mathcal{A}$ will denote the projective ring ${}^h\mathcal{A} = K[x_1, \cdots, x_n; y]$, which results from the introduction of y.

To return from ${}^h\mathcal{A}$ to the original ring, use the natural homomorphism ${}^a p$ defined by partially evaluating p at $y = 1$. For example, ${}^h(x_1^3 + x_2) = x_1^3 + x_2 y^2$, and ${}^a(x_2^2 y + 3x_1 x_3^2) = x_2^2 + 3x_1 x_3^2$.

2.4. Hilbert functions. The Hilbert function of a homogeneous set T is denoted by $\varphi_T(z)$ and is defined as follows:

$$\varphi_T(z) = \text{the dimension of } T_z \text{ as a vector space over } K.$$

Equivalently, let $\underset{A}{>}$ be any fixed admissible ordering. The Hilbert function may be defined to be the number of degree z power products that occur as the head monomial of a polynomial of T. That is,

$$\varphi_T(z) = |\{p \in \mathrm{PP}[X] : p \in \mathrm{Head}_A(T_z)\}| .$$

The definitions of homogeneous direct decompositions and Hilbert functions lead immediately to Lemma 2.2.

LEMMA 2.2. *Let $S_1, \cdots, S_r$ be a homogeneous direct decomposition of T; then* $\varphi_T(z) = \sum_{i=1}^{r} \varphi_{S_i}(z)$.

Let $I \subseteq {}^h\mathcal{A}$ be a homogeneous ideal. If N is any homogeneous set of representatives for the quotient ring ${}^h\mathcal{A}/I$, then $I \oplus N$ is a homogeneous direct decomposition for the entire ring ${}^h\mathcal{A}$. Therefore the Hilbert function of N (and hence ${}^h\mathcal{A}/I$) satisfies the relation

$$\varphi_{{}^h\mathcal{A}/I}(z) = \varphi_N(z) = \varphi_{{}^h\mathcal{A}}(z) - \varphi_I(z) .$$

In particular, since I is a homogeneous ideal, a homogeneous system of representatives for the ring ${}^h\mathcal{A}/I$ can be constructed as

$$N_I = \{\mathrm{nf}_G(a) : a \in {}^h\mathcal{A}\} ,$$

where G is any Gröbner basis for I.

It is a classic result that for any ideal I, at sufficiently large z, the Hilbert functions $\varphi_I(z)$ and $\varphi_{{}^h\mathcal{A}/I}(z)$ become polynomials in z. These polynomials will be denoted using the notation $\overline{\varphi}_I(z)$ and $\overline{\varphi}_{{}^h\mathcal{A}/I}(z)$.

3. Cone decompositions of the polynomial ring. The main goal in finding a direct decomposition for an ideal I is to partition I into subsets whose Hilbert function can easily be described. In particular, the types of elements desired are sets of the form $\{ah : a \in K[u]\}$, where h is a homogeneous polynomial and u is a subset of $X = \{x_1, \cdots, x_n\}$.

DEFINITION. For h a homogeneous polynomial and $u \subseteq X$, the set $\{ah : a \in K[u]\}$ is called a *cone* and is denoted by $C(h, u)$.

Some insight into the behavior of cones can be gained from considering monomial ideals with $n = 2$. This case can be well understood because the cones may be depicted graphically. However, since many interesting phenomena occur only at the higher dimensions this simple case can at times be misleading. For example, in two dimensions all Borel-fixed ideals are lexicographic. Furthermore, there are many features of general polynomial ideals that do not appear in the monomial case. This problem is not so important here though, because the cone decomposition will be applied mainly to monomial ideals.

The graphical representation in two dimensions is illustrated in Fig. 1. The power products are represented in a triangular grid with 1 at the bottom. Powers of x run along the left side of the grid, and powers of y to the right. To reach the vertex associated with a given power product $x^a y^b$ count a places upward to the left and then

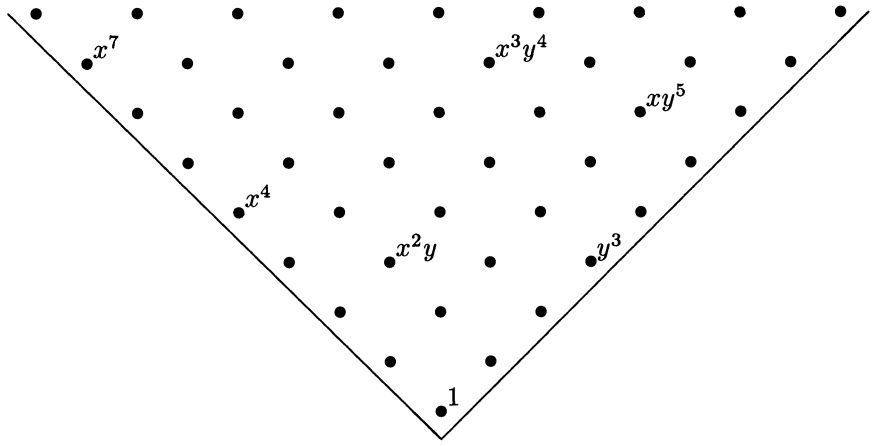


FIG. 1. *The power-products of* $K[x, y]$.

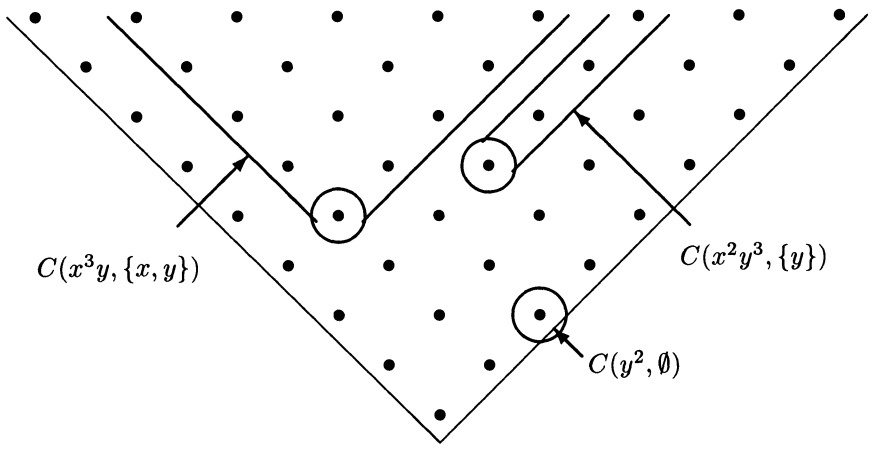


FIG. 2. *Examples of cones.*

b places upward to the right. Figure 2 illustrates that cones may then be represented in the diagram by encircling the power products which the cones contain.

For a cone $C(h,u)$, the Hilbert function of $C(h,u)$ is dependent only on $\deg(h)$ and $|u|$. Counting the number of power products in $\text{Head}(C(h,u))$, we find that if $u = \emptyset$, then

$$\varphi_{C(h,\emptyset)}(z) = \begin{cases} 0, & z \neq \deg(h), \\ 1, & z = \deg(h), \end{cases}$$

and for $|u| > 0$,

$$\varphi_{C(h,u)}(z) = \begin{cases} 0, & z < \deg(h), \\ \begin{pmatrix} z - \deg(h) + |u| - 1 \\ |u| - 1 \end{pmatrix}, & z \geq \deg(h). \end{cases}$$

DEFINITION. Let $h_1, \cdots, h_r$ be homogeneous polynomials of $\mathcal{A}$, and let $u_1, \cdots, u_r$ be subsets of X. A finite set $P = \{\langle h_1, u_1\rangle, \cdots, \langle h_r, u_r\rangle\}$ is a *cone decomposition* of $T \subseteq \mathcal{A}$ if the cones $C(h_i, u_i)$ form a direct decomposition of T.

The cones $\langle h_i, u_i\rangle \in P$ that have $u_i = \emptyset$ form a finite part of T and do not contribute to the Hilbert polynomial of T. The remaining cones, for which $u_i \neq \emptyset$ form a direct decomposition of a set that is equivalent to T at large degrees. This portion of the cone decomposition will be denoted as

$$P^+ = \{\langle h, u\rangle \in P : u \neq \emptyset\} .$$

A cone decomposition P for T is said to be *k-standard* (k an integer) if the following two conditions hold:

(1) There is no pair $\langle h, u\rangle \in P^+$ with $\deg(h) < k$.

(2) For every $\langle g, v\rangle \in P^+$ and degree d such that $k \leq d \leq \deg(g)$, P contains a pair $\langle h, u\rangle$ with $\deg(h) = d$ and $|u| \geq |v|$.

Note that if P^+ is the empty set, then P is k-standard for all natural numbers k. On the other hand, if P^+ is nonempty, the only possible value for k is $\min\{\deg(h) : \langle h, u\rangle \in P^+\}$.

The following list contains an assortment of easily verifiable properties of cone decompositions and k-standard cone decompositions.

(1) $\emptyset$ is a 0-standard cone decomposition for $\emptyset$.

(2) $\{\langle h, u\rangle\}$ is a $\deg(h)$-standard cone decomposition of $C(h,u)$.

(3) $\{\langle 1, X\rangle\}$ is a 0-standard cone decomposition of $\mathcal{A}$.

(4) Let S_1 and S_2 be a direct decomposition of T, and let P_1 and P_2 be cone decompositions of S_1 and S_2, respectively. Then $P_1 \cup P_2$ is a cone decomposition of T.

(5) Let S_1 and S_2 be a direct decomposition of T, and let P_1 and P_2 be k-standard cone decompositions of S_1 and S_2, respectively. Then $P = P_1 \cup P_2$ is a k-standard cone decomposition of T.

(6) If $P = \{\langle h_1, u_1\rangle, \cdots, \langle h_s, u_s\rangle\}$ is a k-standard cone decomposition for T, then for any homogeneous polynomial c, the set $P' = \{\langle ch_1, u_1\rangle, \cdots, \langle ch_s, u_s\rangle\}$ is a $(k + \deg(c))$-standard cone decomposition for $\{ch : h \in T\}$.

There is one special cone decomposition that provides a useful function for manipulations.

DEFINITION. Let $u = \{x_{j_1}, \cdots, x_{j_m}\} \subseteq X$. Then define the set $E(h,u)$ as

$$E(h,u) = \{\langle h, \emptyset\rangle\} \cup \{\langle x_{j_i} h, \{x_{j_i}, \cdots, x_{j_m}\}\rangle : i = 1, \cdots, m\} .$$

It is easy to verify that $E(h,u)$ is a $(\deg(h)+1)$-standard cone decomposition of $C(h,u)$.

LEMMA 3.1. *Let P be a k-standard cone decomposition for T. Then, for any $d \geq k$, there exists a d-standard cone decomposition P_d for the set T.*

Proof. If $P^+ = \emptyset$, then the result holds trivially, so assume that P^+ is nonempty. It suffices to show that $(k+1)$-standard cone decomposition exists for T. Let $R = \{\langle h,u\rangle \in P : \deg(h) = k\}$, and $S = P - R$. The original set P was k-standard, so after removing the cones in R, the remaining set S is $(k+1)$ standard.

Since R contains only pairs $\langle h,u\rangle$ for which $\deg(h) = k$, R is trivially k-standard. The set spanned by the cones in R also has a $(k+1)$-standard cone decomposition, namely,

$$R' = \bigcup_{\langle h,u\rangle \in R} E(h,u) .$$

Finally, $P_{k+1} = R' \cup S$ is a $(k+1)$-standard cone decomposition for T. □

COROLLARY 3.2. *Let $S_1, \cdots, S_r$ be a direct decomposition of T, where for each S_i there exists a k_i-standard cone decomposition P_i. Then there exists a k-standard cone decomposition P of T with $k = \max\{k_1, \cdots, k_r\}$.*

4. Splitting a system of representatives. In this section it will be shown that for any homogeneous ideal I, it is possible to construct a 0-standard cone decomposition for N_I. Recall that once the ordering $>_A$ and a Gröbner basis G for I are fixed, then N_I and $N_{\mathrm{Head}_A(G)}$ have a termwise agreement as sets. Thus, only monomial ideals need be considered.

Let I be an ideal of $\mathcal{A}$ generated by the set of monomials $F = \{f_1, \cdots, f_r\}$. For a given variable x_j, there is a direct decomposition of I consisting of I_0 and I_1, where

$$I_0 = I \cap K[X - \{x_j\}]$$

and,

$$I_1 = I \cap (x_j) = \{x_j h : h \in \mathcal{A} \text{ and } x_j h \in I\} .$$

Clearly, I_0 is an ideal of $K[X - \{x_j\}]$ generated by $F \cap K[X - \{x_j\}]$. It is also easy to verify that I_1 is an ideal of $\mathcal{A}$ generated by the set $G = \{g_1, \cdots, g_r\}$, where

$$g_i = \begin{cases} x_j f_i, & f_i \in K[X - \{x_j\}], \\ f_i & \text{otherwise.} \end{cases}$$

Comparing the ideal I_1 defined above with the quotient $I : x_j$, shows that $I_1 = \{x_j h : h \in I : x_j\}$. Furthermore, this leads to the fact that $I : x_j$ is generated by $H = \{h_1, \cdots, h_r\}$ where

$$h_i = x_j^{-1} g_i = \begin{cases} f_i, & f_i \in K[X - \{x_j\}], \\ x_j^{-1} f_i & \text{otherwise.} \end{cases}$$

This method of forming a basis for $I : x_j$ is restated as an algorithm in Fig. 3.

DEFINITION. Let $P \cup Q$ be a cone decomposition of $T \subseteq \mathcal{A}$, and let I be an ideal of $\mathcal{A}$. Then P and Q are said to *split T relative to I* if $\langle h,u\rangle \in P$ implies $C(h,u) \subseteq I$ (i.e., $h \in I$), and $\langle h,u\rangle \in Q$ implies $C(h,u) \cap I = \{0\}$. We may easily verify that P is a cone decomposition of $T \cap I$. Furthermore, the following lemma shows that under proper restrictions Q is a cone decomposition for $T \cap N_I$.

```
QUOTIENT_BASIS(F, x_j)
    Input :   F a monomial basis for I ⊆ A
              x_j ∈ X a variable
    Output :  F' a monomial basis for I : x_j.

    F' := ∅
    For f_i ∈ F
        if f_i ∈ K[X − {x_j}] then F' := F' ∪ {f_i}
                             else F' := F' ∪ {x_j^{-1} f_i}
    return (F')
End.
```

FIG. 3. *The algorithm for forming a basis for* $I : x_j$.

LEMMA 4.1. *Let* $P = \{\langle g_1, u_1\rangle, \cdots, \langle g_r, u_r\rangle\}$ *and* $Q = \{\langle h_1, v_1\rangle, \cdots, \langle h_s, v_s\rangle\}$ *split* T *relative to a monomial ideal* I, *where for each* $\langle h_i, v_i\rangle \in Q$, h_i *is a monomial. Then* Q *is a cone decomposition for* $T \cap N_I$.

Proof. If I is a monomial ideal, then regardless of admissible ordering $>_A$ and Gröbner basis G,

$$f \in N_I \iff \text{each monomial of } f \text{ is not in } I.$$

Furthermore, if h_i is a monomial then

$$f \in C(h_i, v_i) \iff \text{each monomial of } f \text{ is in } C(h_i, v_i).$$

By the definition of a *splitting* set of cones, $C(h_i, v_i) \cap I = \{0\}$, so

$$f \in C(h_i, v_i) \iff \text{no monomial of } f \text{ is in } I.$$

Let $f \in T \cap N_I$. $f \in T$ implies that f can be written uniquely as

$$f = f_{P_1} + f_{P_2} + \cdots + f_{P_r} + f_{Q_1} + \cdots + f_{Q_s}\ .$$

The partial sum $f_{\overline{P}} = f_{P_1} + f_{P_2} + \cdots + f_{P_r}$ is in I and therefore is a sum of monomials in I. Since no such monomials appear in the cones $C(h_i, v_i)$, all monomials of $f_{\overline{P}}$ must also appear in f. But $f \in N_I$ and can include no monomial of I. Hence, $f_{\overline{P}} = 0$ and f can be written uniquely as $f = f_{Q_1} + \cdots + f_{Q_s}$. □

As an example, consider the monomial ideal $I = (x^4y, xy^3, y^5)$. This ideal has a cone decomposition given by the set

$$\{C(x^4y, \{x\}),\ C(x^4y^2, \{x\}),\ C(xy^3, \{x, y\}),\ C(y^5, \{y\})\}\ .$$

This particular cone decomposition is illustrated in Fig. 4. Viewing this figure should make it clear that this cone decomposition is not unique.

For example, the cones $C(xy^3, \{x, y\})$ and $C(y^5, \{y\})$ can be equivalently replaced by $C(xy^3, \{x\})$, $C(xy^4, \{x\})$, and $C(y^5, \{x, y\})$.

Now, let T be the ideal generated by x^2. Then $T \cap I$ has a cone decomposition described by the set

$$P = \{\langle x^4y, \{x\}\rangle, \langle x^4y^2, \{x\}\rangle, \langle x^2y^3, \{x, y\}\rangle\}\ .$$

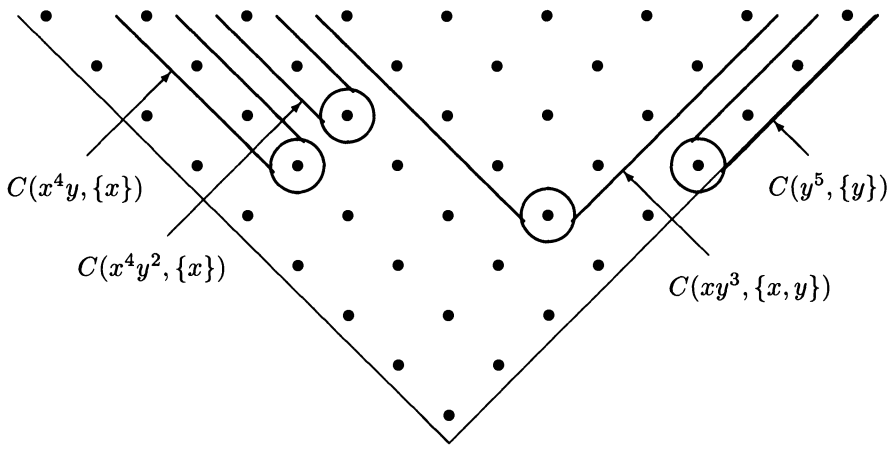


FIG. 4. *The cone decomposition of* $I = (x^4y, xy^3, y^5)$.

This cone decomposition is illustrated in Fig. 5.

But *splitting* I requires not only a cone decomposition P for the ideal, but also a cone decomposition Q for $T \cap N_I$. In this example, Fig. 6 shows that Q may be chosen as the set of cones described by

$$Q = \{\langle x^2, \{x\}\rangle, \langle x^2y, \emptyset\rangle, \langle x^2y^2, \emptyset\rangle, \langle x^3y, \emptyset\rangle, \langle x^3y^2, \emptyset\rangle\} .$$

From the definition of what is meant by a cone decomposition $P \cup Q$ *splitting* a set relative to an ideal I, it is immediate that a cone $C(h, u)$ can belong to such a decomposition only if either $C(h, u) \subseteq I$ or $C(h, u) \cap I = \emptyset$. The following lemma shows that if the ideal I is a monomial ideal and h is also a monomial, then this condition can be effectively determined. This will provide an algorithm to split the ring $\mathcal{A}$ relative to a monomial ideal I.

LEMMA 4.2. *Let* I *be a monomial ideal,* $h \in \mathrm{PP}[X]$, $u \subset X$, *and let* F *be a power product basis for* $I : h$. *Then,*

(1) $C(h, u) \subseteq I$ *if and only if* $1 \in F$.

(2) $C(h, u) \cap I = \emptyset$ *if and only if* $F \cap \mathrm{PP}[u] = \emptyset$.

Proof. (1) $1 \in F \iff 1 \in I : h \iff h \in I \iff C(h, X) \subseteq I$.

(2) ($\Longrightarrow$) Assume $C(h, u) \cap I = \emptyset$. Then for $g \in \mathrm{PP}[u]$,

$$\begin{aligned} hg \in C(h, u) &\implies hg \notin I \\ &\implies g \notin I : h \\ &\implies g \notin F . \end{aligned}$$

(2) ($\Longleftarrow$) Assume $F \cap \mathrm{PP}[u] = \emptyset$. Then for $g \in \mathrm{PP}[u]$, g cannot be in $I : h$ since otherwise F would have to contain a divisor of g and this divisor would also be in $\mathrm{PP}[u]$. So

$$g \notin I : h \implies hg \notin I .$$

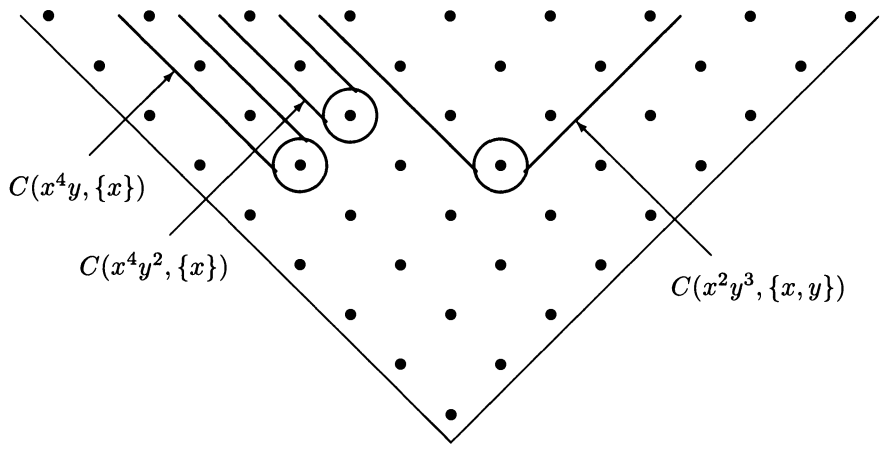


FIG. 5. *The cone decomposition of* $(x^2) \cap I$.

By the definition of $C(h, u)$, every polynomial contained in this cone is of the form hg with $g \in \mathrm{PP}[u]$ and hence not in the ideal I. □

Figure 7 provides an algorithm SPLIT for splitting a cone $C(h, u)$ with respect to a monomial ideal I.

LEMMA 4.3. *The algorithm* SPLIT *terminates.*

Proof. For a set of arguments h, u, and F, define the rank of the arguments as $|u| + \sum_{f \in F} \deg(f)$. It is now claimed that if SPLIT is invoked with arguments of rank r, then the two recursive calls (if reached) have arguments of rank $\leq r - 1$. For the first call, this is trivial. For the second call, it must be shown that there is some $f_i \in F$ such that $f_i \notin \mathrm{PP}[X - \{x_j\}]$. But, this must be true since otherwise $F \cap K[s \cup \{x_j\}] = \emptyset$, contradicting the choice of s, and hence x_j.

If $r = 0$, then F must either be $\{1\}$, or $\emptyset$. In either case, the recursion stops. Therefore the depth of recursion is at most r, and hence the algorithm terminates. □

LEMMA 4.4. *The algorithm* SPLIT *is correct.*

Proof. The previous lemma assures the termination of the algorithm, so the correctness of the algorithm can be proven using induction on the depth of recursion.

The basis case in which no recursive calls are made occurs if $1 \in F$ or $F \cap \mathrm{PP}[u] = \emptyset$. In both of these cases, Lemma 4.2 shows that the trivial decomposition (P, Q) satisfies the definition for splitting $C(h, u)$ relative to I.

Otherwise, the cone $C(h, u)$ is decomposed into

$$C(h, u) \;=\; C(h, u - \{x_j\}) \oplus C(x_j h, u) \;.$$

Since F is a power product basis for $I : h$, the function QUOTIENT_BASIS produces a power product basis F' for the ideal $I : x_j h$. Inductively, the algorithm SPLIT returns

(1) (P_0, Q_0), which splits $C(h, u - \{x_j\})$ relative to I, and

(2) (P_1, Q_1), which splits $C(x_j h, u)$ relative to I.

These two decompositions are then joined to produce the desired decomposition of $C(h, u)$. □

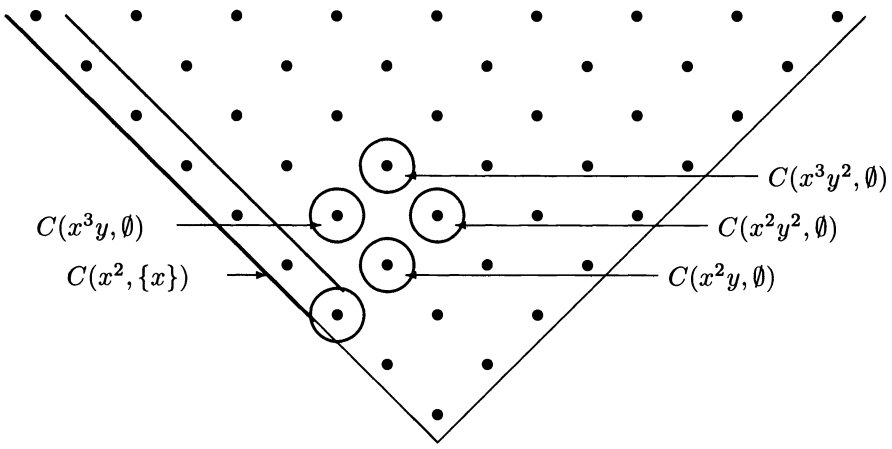


FIG. 6. *The cone decomposition of* $(x^2) \cap N_I$.

In the SPLIT algorithm, the choice of $s \subset u$ such that $F \cap \mathrm{PP}[s] = \emptyset$ as a *maximal* subset is not a necessary condition for the correctness of the algorithm in producing a splitting decomposition. However, it will soon be shown that the set Q returned by this algorithm has the additional property of being $\deg(h)$-standard. To prove that this is indeed true, we begin with a simple lemma regarding the condition $F \cap \mathrm{PP}[s] = \emptyset$.

LEMMA 4.5. *Let* h, u, I, *and* F *be as in algorithm* SPLIT. *Then for any* $v \subseteq X$,

$$C(h,v) \subseteq C(h,u) \cap N_I \iff v \subseteq u \text{ and } F \cap \mathrm{PP}[v] = \emptyset \ .$$

Proof. ($\Longrightarrow$) $C(h,v) \subseteq C(h,u)$ clearly implies $v \subseteq u$. To see that $F \cap \mathrm{PP}[v] = \emptyset$, let f be any nonzero element of $K[v]$. Then $hf \in C(h,v) \subseteq N_I$. But $I \cap N_I = \{0\}$ and neither h nor f is zero, so $hf \notin I$. Then,

$$\begin{aligned} hf \notin I &\implies f \notin I : h \\ &\implies f \notin F \ . \end{aligned}$$

($\Longleftarrow$) $v \subseteq u$ implies $C(h,v) \subseteq C(h,u)$, so it only remains to be shown that $C(h,v) \subseteq N_I$. To prove this, it is sufficient to show that no monomial of $C(h,v)$ belongs to I. Each monomial of $C(h,v)$ is of the form hf with f a monomial of $K[v]$. Then,

$$\begin{aligned} F \cap \mathrm{PP}[v] = \emptyset &\implies f \notin I : h \\ &\implies hf \notin I \ . \end{aligned}$$

□

LEMMA 4.6. *Let* h, u, I, *and* F *be valid input for algorithm* SPLIT, *and let* (P,Q) *denote the sets returned by* SPLIT(h,u,F). *Then for any power product* g, $C(g,v) \subseteq C(h,u) \cap N_I$ *implies that* Q *contains a pair* $\langle h, s\rangle$ *with* $|s| \geq |v|$.

Proof. Using the previous lemma,

$$\begin{aligned} C(g,v) \subseteq C(h,u) \cap N_I &\implies C(h,v) \subseteq C(h,u) \cap N_I \\ &\implies v \subseteq u \text{ and } F \cap \mathrm{PP}[v] = \emptyset. \end{aligned}$$

```
SPLIT(h,u,F)
  Input :  h ∈ PP[X]
           u ⊆ X is a set of variables
           F a power product basis for I : h
  Output : (P,Q) which split C(h,u) relative to I.

  If 1 ∈ F then return (P = {⟨h,u⟩}, Q = ∅)
  If F ∩ PP[u] = ∅ then return (P = ∅, Q = {⟨h,u⟩})
  Otherwise
    Choose s ⊂ u a maximal subset such that F ∩ PP[s] = ∅
    Choose x_j ∈ u − s   [If s=u this point would not be reached.]

    (P_0, Q_0) := SPLIT(h, u − {x_j}, F)

    F' := QUOTIENT_BASIS(F, x_j)
    (P_1, Q_1) := SPLIT(x_j h, u, F')
    return (P = P_0 ∪ P_1, Q = Q_0 ∪ Q_1)
End.
```

FIG. 7. *The algorithm for splitting $C(h,u)$ relative to I.*

We proceed inductively on $|u|-|v|$. If $v = u$, then the algorithm returns $Q = \{\langle h,u\rangle\}$, satisfying the lemma. Otherwise, the choice of s as a *maximal* subset such that $F \cap \mathrm{PP}[s] = \emptyset$ implies that $|s| \geq |v|$. The previous lemma can now be applied in the opposite direction to get

$$C(h,s) \subseteq C(h,u-\{x_j\}) \cap N_I \ .$$

Using the induction hypothesis, the set Q_0 formed by the recursive call $\mathtt{SPLIT}(h,u-\{x_j\},F)$ contains a pair $\langle h,w\rangle$ with $|w| \geq |s| \geq |v|$. The lemma then follows from the fact that Q_0 is a subset of Q. □

A basis $R = \{f_1,\cdots,f_k\}$ for an ideal I is called a *reduced basis* if each f_i satisfies $f_i \notin (R-\{f_i\})$. On the other hand, suppose that F is not a reduced basis, and that $f_i \in (F-\{f_i\})$. Then, $F-\{f_i\}$ is also a basis for I. Successively removing redundant generators produces a subset of F that is a reduced basis for I.

LEMMA 4.7. *Let R be a reduced power product basis for a monomial ideal I, and let $P = \{\langle h_1,u_1\rangle,\cdots,\langle h_r,u_r\rangle\}$ be any cone decomposition of I where the h_i's are power products. Then, for each $f \in R$, there is a pair $\langle f,u\rangle \in P$.*

Proof. Let f be any element of R. Since $f \in I$, there is some $\langle h,u\rangle \in P$ such that $f \in C(h,u)$. But, now h is also in I, so $h = bg$ for some $g \in R$. But $f \in C(h,u)$, so f can be written as $f = ah = abg$. Since R is reduced, we have $ab = 1$, and $f = h$. □

LEMMA 4.8. *Let F be a power product basis for $I \neq \mathcal{A}$, $(P,Q) = \mathtt{SPLIT}(1,X,F)$, and let $R \subseteq F$ be a reduced basis for I. Then for every $f \in R$, Q contains a pair $\langle h,u\rangle$ with* $\deg(h) = \deg(f) - 1$.

Proof. Let f be any element of R. By the preceding lemma there is a pair $\langle f,v\rangle \in P$. Consider how this pair got into P. Since $\deg(f) > 0$, there must have been a recursive call $\mathtt{SPLIT}(f,v,F')$, where F' is a basis for $I : f$. This invocation of SPLIT must have been the child of either

(1) $\mathtt{SPLIT}(x_j^{-1}f,v,F'')$, or

(2) $\texttt{SPLIT}(f, v \cup \{x_j\}, F')$.

Using the first possibility as a basis case, inductively, we may step backward through the computation of $\texttt{SPLIT}(1, X, F)$ to find an invocation $\texttt{SPLIT}(x_j^{-1}f, v', F'')$ with $v' \supseteq v$.

The cone $C(x_j^{-1}f, v')$ could not have been a subset of I, since then a recursive call would not have been generated. Therefore, if

$$(P', Q') \;=\; \texttt{SPLIT}(x_j^{-1}f, v', F''),$$

then Q' is nonempty. But then Lemma 4.6 assures that Q' contains a pair of the form $\langle x_j^{-1}f, s\rangle$. Since $\deg(x_j^{-1}f) = \deg(f) - 1$, the existence of this pair in $Q' \subseteq Q$ satisfies the lemma. □

COROLLARY 4.9. *Let F be a power product basis for I, and let $(P, Q) = \texttt{SPLIT}(1, X, F)$. Then if $d = 1 + \max\{\deg(h) : \langle h, u\rangle \in Q\}$, I can be generated by the set $\{f \in F : \deg(f) \leq d\}$.*

LEMMA 4.10. *Let $(P, Q) = \texttt{SPLIT}(h, u, F)$. Q is a $\deg(h)$-standard cone decomposition.*

Proof. If Q is either $\emptyset$ or $\{\langle h, u\rangle\}$, then the lemma follows trivially. Otherwise, assume inductively (on the number of recursions) that Q_0 and Q_1 satisfy the lemma. That is Q_0 and Q_1 are, respectively, $\deg(h)$-standard and $(\deg(h) + 1)$-standard.

To show that Q is a $\deg(h)$-standard cone decomposition, it must be shown that for any $\langle g, v\rangle \in Q$ and degree d such that $\deg(h) \leq d \leq \deg(g)$, there is a pair $\langle p, t\rangle \in Q$ with $\deg(p) = d$ and $|t| \geq |v|$. Since $Q = Q_0 \cup Q_1$, there are two cases to consider.

(1) $\langle g, v\rangle \in Q_0$. Since Q_0 is itself a $\deg(h)$-standard cone decomposition, Q_0 contains all the pairs needed to satisfy the condition for $\langle g, v\rangle$, and Q_0 is a subset of Q.

(2) $\langle g, v\rangle \in Q_1$. Since Q_1 is a $(\deg(h) + 1)$-standard cone decomposition, Q_1 contains the pairs needed to satisfy the condition for $\langle g, v\rangle$ for $\deg(h)+1 \leq d \leq \deg(g)$. For $d = \deg(h)$, Lemma 4.6 assures that Q contains the needed pair. □

The remarks at the beginning of this section allow these results to be extended beyond monomial ideals.

THEOREM 4.11. *Let G be a Gröbner basis for I with respect to $>_A$. Let $(P, Q) = \texttt{SPLIT}(1, X, \mathrm{Hterm}_A(G))$. Then Q is a 0-standard cone decomposition of $N_I = \mathrm{nf}_G(\mathcal{A})$. Furthermore, if $d = 1 + \max\{\deg h : \langle h, u\rangle \in Q\}$, then $G' = \{g \in G : \deg(g) \leq d\}$ is also a Gröbner basis for I with respect to $>_A$.*

Proof. $\mathrm{Hterm}_A(G)$ is a basis for $\mathrm{in}_A(I)$. Therefore the SPLIT algorithm returns a 0-standard cone decomposition for $N_{\mathrm{in}_A(I)} = N_I$.

By Corollary 4.9, the set

$$\{h \in \mathrm{Hterm}_A(G) : \deg(h) \leq d\} \;\subseteq\; \mathrm{Hterm}_A(G')$$

is a basis for $\mathrm{in}_A(I)$, and hence G' is a Gröbner basis for I. □

5. Splitting a homogeneous ideal. So far we have seen that for any ideal I, there exists a 0-standard cone decomposition of N_I. But what about I itself? The construction SPLIT provides a cone decomposition of I that is only valid for monomial ideals, and even this does not produce a standard cone decomposition. The answer is found in the following lemma.

LEMMA 5.1. *Let $F = \{f_1, \cdots, f_r\}$ be a homogeneous basis for an ideal I; then there exists a k-standard cone decomposition P for I with*

$$k = \max\{\deg(f_i) \ : \ i = 1 \cdots r\}.$$

Proof. Let $S_1 = (f_1)$, and for $i = 2 \cdots r$ let $J_i = (f_1, \cdots, f_{i-1})$, $L_i = J_i : f_i$ and $S_i = \{cf_i \ : \ c \in N_{L_i}\}$. The sets $S_1, \cdots, S_r$ form a direct decomposition of I. S_1 is a principal ideal that has the $\deg(f_1)$-standard cone decomposition $P_1 = \{\langle f_1, X\rangle\}$. Using the construction provided by SPLIT, we form a 0-standard cone decomposition Q_i for each N_{L_i}. If $Q_i = \{\langle h_1, u_1\rangle, \cdots, \langle h_s, u_s\rangle\}$, then it follows that $P_i = \{\langle f_r h_1, u_1\rangle, \cdots, \langle f_r h_s, u_s\rangle\}$ is a $\deg(f_i)$-standard cone decomposition for S_i.

It then follows from Corollary 3.2 that there exists a k-standard cone decomposition P for I. □

For $I \neq \{0\}$, it will be preferable to use a slightly modified version of this result. When the sets P_i are united to form P, do not include P_1 in the union. This produces the following modified result.

COROLLARY 5.2. *Let $F = \{f_1, \cdots, f_r\}$ be a homogeneous basis for an ideal I with $r > 0$, and let $S_1, \cdots, S_r$ be as above. Then there exists a direct decomposition of I consisting of the primary ideal $S_1 = (f_1)$, and a k-standard cone decomposition P for $S_2 \oplus S_3 \oplus \cdots \oplus S_r$ with*

$$k = \max\{\deg(f_i) \ : \ i = 1 \cdots r\}.$$

6. The exact cone decomposition.

DEFINITION. For $T \subseteq K[X]$, Q is called an *exact cone decomposition* of T if Q is a k-standard cone decomposition of T for some k, and additionally for every degree d, Q^+ contains at most one pair $\langle h, u\rangle$ with $\deg(h) = d$.

If Q^+ is nonempty, then there is a unique value of k for which Q is k-standard. Let $\overline{a}_Q$ denote this value of k. In the case that Q^+ is empty, let $\overline{a}_Q = 0$. Both of these cases can be captured with the single definition: $\overline{a}_q$ is the least value of k such that Q is k-standard. However, this unified definition fails to emphasize the fact that in the more important case ($Q^+ \neq \emptyset$) the value of k is unique.

For $i = 0, \cdots, n+1$, let

$$b_i = \min\{d \geq \overline{a}_Q \ : \ \langle h, u\rangle \in Q \text{ and } |u| \geq i \implies \deg(h) < d\} \ .$$

It is a simple consequence of this definition that the b_i's satisfy $b_0 \geq b_1 \geq \cdots \geq b_{n+1} = \overline{a}_Q$. Furthermore,

$$b_1 \ = \ \begin{cases} 1 + \max\{\deg(h) \ : \ \langle h, u\rangle \in Q^+\}, & Q^+ \neq \emptyset, \\ 0, & Q^+ = \emptyset \ . \end{cases}$$

LEMMA 6.1. *Let Q be an exact cone decomposition, and let $b_0, \cdots, b_{n+1}$ be defined as above. Then for each $i = 1, \cdots, n$ and degree d such that $b_{i+1} \leq d < b_i$, there is exactly one pair $\langle h, u\rangle \in Q^+$ such that $\deg(h) = d$ and in that pair $|u| = i$.*

Proof. If Q^+ is empty, then $b_1 = b_2 = \cdots = b_{n+1} = 0$ and the lemma follows vacuously. Otherwise, for each $i = 1, \cdots, n$, the definition of b_i requires that $b_1 - 1$ be the largest degree such that Q contains a pair $\langle g, v\rangle$ with $|v| \geq i$. Since Q is b_{n+1}-standard, each degree $d = b_{n+1}, \cdots, b_i - 1$ must have a pair $\langle h_d, u_d\rangle \in Q$ with

$\deg(h_d) = d$ and $|u_d| \geq |v| \geq i$. Since Q is an exact cone decomposition $\langle h_d, u_d \rangle$ is the only pair $\langle h, u \rangle \in Q^+$ with $\deg(h) = d$.

Now if $b_i = b_{i+1}$ the range $d = b_{i+1}, \cdots, b_i - 1$ is vacuous. Otherwise, for each d in this range $|u_d| = i$ since $|u_d| > i$ would contradict the definition of b_{i+1}. □

The following trivial lemma provides a tool by which any standard cone decomposition may be transformed into an exact cone decomposition.

LEMMA 6.2. *Let Q be a k-standard cone decomposition of T, and let $\langle f, s \rangle$, $\langle g, v \rangle \in Q$ such that $\deg(f) = \deg(g)$, and $|v| \geq |s| > 0$. Then for any $x_j \in s$,*

$$Q' = (Q - \{\langle f, s \rangle\}) \cup \{\langle f, s - \{x_j\} \rangle, \langle x_j f, s \rangle\}$$

is also a k-standard cone decomposition of T.

Proof. It must be shown that for every pair $\langle \ell, w \rangle \in Q'$ and degree $d = k, \cdots, \deg(\ell)$ there is a pair $\langle h, u \rangle \in Q'$ with $\deg(h) = d$ and $|u| \geq |w|$. For $\langle \ell, w \rangle \in Q \cap Q'$, Q' *inherits* all the required pairs from Q. For the two new pairs, the presence of $\langle g, v \rangle \in Q'$ is sufficient to show that Q' must again contain the required pairs. □

This lemma provides a tool to *shift* pairs away from degrees occupied by other pairs.

One new term will be introduced only for the purposes of proving the correctness of the following algorithm. A k-standard cone decomposition P is called *m-exact* if for each degree d there is at most one pair $\langle h, u \rangle \in P$ such that $\deg(h) = d$ and $|u| > m$. With this definition, a cone decomposition is exact if and only if it is 0-exact. It also follows vacuously that any cone decomposition is n-exact. Consider the algorithm of Fig. 8.

```
SHIFT(Q, k, m)
    Input :   Q a k-standard m-exact cone decomposition for T
    Output :  Q' a k-standard (m − 1)-exact cone decomposition for T.
    Q' := Q
    If {⟨h, u⟩ ∈ Q : |u| ≥ m} = ∅ then return(Q').
    c := |{⟨h, u⟩ ∈ Q : |u| ≥ m}|.
    For d := k to k + c − 1 do
      B := {⟨h, u⟩ ∈ Q' : deg(h) = d and |u| ≥ m}
      While |B| > 1 loop
            Choose ⟨h, u⟩ ∈ B with |u| = m
            Choose x_j ∈ u
            B := B − {⟨h, u⟩}
            Q' := (Q' − {⟨h, u⟩}) ∪ {⟨h, u − {x_j}⟩, ⟨x_j h, u⟩}
      End While loop
    End For d loop
    return(Q')
End.
```

FIG. 8. *The algorithm for shifting pairs in a standard cone decomposition.*

LEMMA 6.3. *The algorithm* SHIFT *is correct.*

Proof. If Q is a k-standard partition, then it follows from the previous lemma that the set Q' will also be k-standard. Furthermore, the action of the algorithm assures that for each degree $d < k + c$, Q' will contain at most one pair $\langle h, u \rangle$ with $\deg(h) = d$ and $|u| \geq m$. But what about degrees $\geq k + c$? Checking the line at which Q' is modified will show that throughout the execution of this algorithm the

size of the set $\{\langle h,u\rangle \in Q' : |u| \geq m\}$ remains invariantly c. Now since Q' is k-standard, a pair $\langle g,v\rangle \in Q'$ with $|v| \geq m$ requires that Q' contain a pair $\langle h_d,u_d\rangle$ with $|u_d| \geq m$ and $\deg(h_d) = d$, for every degree $d = k,\cdots,\deg(g)$. The c pairs in the set $\{\langle h,u\rangle \in Q' : |u| \geq m\}$ must then include the $\deg(g) - k + 1$ pairs of the form $\langle h_d,u_d\rangle$. Therefore, $\deg(g) \leq c + k - 1$. □

Now, the SHIFT algorithm can be used to produce an exact cone decomposition using the algorithm in Fig. 9. Note that the action of the EXACT and SHIFT algorithms

```
EXACT(Q, k)
    Input :   Q a k-standard cone decomposition for T
    Output :  Q' an exact cone decomposition for T.
    Q_n := Q
    For m := n down to 1 do
        Q_{m-1} := SHIFT(Q_m, k, m)
    End For m loop
    return(Q_0)
End.
```

FIG. 9. *The algorithm for producing an exact partition.*

assures that if Q' is the exact cone decomposition produced by EXACT(Q,k), then the Macaulay constant b_0 for Q' satisfies

$$b_0 \geq 1 + \max\{\deg(h) : \langle h,u\rangle \in Q\}.$$

7. Exact cone decomposition and Hilbert function. For any cone decomposition P of a set T, the Hilbert function of T can be described by summing the Hilbert functions of the cones in P:

$$\varphi_T(z) = \sum_{\langle h,u\rangle \in P} \varphi_{C(h,u)}(z) .$$

For degrees z greater than or equal to

$$z' = \max\{\deg(h) : \langle h,u\rangle \in P\}$$

each of the cones has a Hilbert function described by the binomial coefficient

$$\overline{\varphi}_{C(h,u)} = \binom{z - \deg(h) + |u| - 1}{|u| - 1},$$

and so

$$\overline{\varphi}_T(z) = \sum_{\langle h,u\rangle \in P^+} \binom{z - \deg(h) + |u| - 1}{|u| - 1} .$$

But, if P is exact, then the constants $b_1,\cdots,b_{n+1}$ describe all of the cones in P^+, so

$$\overline{\varphi}_T(z) = \sum_{j=1}^{n} \sum_{d=b_{j+1}}^{b_j - 1} \binom{z - d + j - 1}{j - 1} .$$

Furthermore, the constant b_0 is defined to be the same as the constant z' given above, so the Hilbert function attains this polynomial form for degrees $z \geq b_0$.

Using the combinatorial identity

$$\sum_{d=b_{j+1}}^{b_j-1} \binom{z-d+j-1}{j-1} = \binom{z-b_{j+1}+j}{j} - \binom{z-b_j+j}{j},$$

the Hilbert function of T can be written in the form:

$$\begin{aligned}
\varphi_T(z) &= \sum_{j=1}^{n} \left[\binom{z-b_{j+1}+j}{j} - \binom{z-b_j+j}{j} \right] \\
&= \binom{z-b_{n+1}+n}{n} - \binom{z-b_1+1}{1} \\
&\quad + \sum_{j=1}^{n-1} \left[\binom{z-b_{j+1}+j}{j} - \binom{z-b_{j+1}+j+1}{j+1} \right] \\
&= \binom{z-b_{n+1}+n}{n} - 1 - \binom{z-b_1}{1} - \sum_{j=1}^{n-1} \binom{z-b_{j+1}+j}{j+1} \\
&= \binom{z-b_{n+1}+n}{n} - 1 - \sum_{j=0}^{n-1} \binom{z-b_{j+1}+j}{j+1}.
\end{aligned}$$

Replacing the summation variable with $i = j + 1$, this can be restated as

$$(*) \qquad \varphi_T(z) = \binom{z-b_{n+1}+n}{n} - 1 - \sum_{i=1}^{n} \binom{z-b_i+i-1}{i}.$$

In the classic paper [7], Macaulay first proved that for sufficiently high degree z, the Hilbert function of a polynomial quotient ring always attains the form of a polynomial such as the one given in (*). For this reason the constants $b_0, \cdots, b_{n+1}$ will be referred to as the Macaulay constants of T. The formulation given above has the added benefit of the additional constant b_0, which provides a bound on the point at which the Hilbert function $\varphi_T(z)$ attains its polynomial form $\overline{\varphi}_T(z)$ as given in (*).

For z in the range $b_1 \leq z < b_0$, the Hilbert functions of the cones in P^+ attain the polynomial forms used in calculating $\overline{\varphi}_T(z)$. For z in this range however, there are also some cones $C(h, \emptyset) - \in P - P^+$, which contribute to the Hilbert function of T. Therefore, for $z \geq b_1$ the following form of the Hilbert function is valid:

$$\begin{aligned}
\varphi_T(z) &= \overline{\varphi}_T(z) + \sum_{\langle h, \emptyset \rangle \in P} \varphi_{C(h,\emptyset)}(z) \\
&= \overline{\varphi}_T(z) + |\{\langle h, \emptyset \rangle \in P : \deg(h) = z\}| .
\end{aligned}$$

LEMMA 7.1. *Let P be any exact cone decomposition for a set T. Once the constant $b_{n+1} = \overline{a}_Q$ is fixed, the constants $b_0, b_1, \cdots, b_n$ are uniquely determined.*

Proof. The Hilbert polynomial of T can be written in the form

$$\overline{\varphi}_T(z) = a_{n-1}z^{n-1} + a_{n-2}z^{n-2} + \cdots + a_1 z + a_0 .$$

Assume inductively that the constants $b_{j+1}, \cdots, b_{n+1}$ have been uniquely determined such that Hilbert function given by (*) agrees with the coefficients $a_{n-1}, \cdots, a_j$. The binomial coefficient

$$\binom{z - b_i + i - 1}{i}$$

is a degree i monic polynomial in z. So, the coefficients $b_{j-1}, \cdots, b_1$ do not effect the coefficient of z^{j-1} in the Hilbert polynomial (*). Therefore, matching the coefficient a_{j-1} requires a unique choice for b_j.

We may then also uniquely determine b_0 as

$$b_0 = \min\{d \geq b_1 : \forall_{z>d} \overline{\varphi}_T(z) = \varphi_T(z)\} . \qquad \square$$

LEMMA 7.2. *Let I be a homogeneous ideal, then the Hilbert function of N_I is described by a unique set of Macaulay constants $b_0 \geq b_1 \geq \cdots \geq b_{n+1} = 0$. Furthermore, for any admissible ordering $\underset{A}{>}$ the degree of polynomials in a reduced Gröbner basis for I with respect to $\underset{A}{>}$ is bounded by b_0.*

Proof. Since N_I has a 0-standard cone decomposition, it is possible to find an exact cone decomposition for N_I with $b_{n+1} = 0$. Once b_{n+1} is fixed as zero, the other Macaulay constants are uniquely determined.

Let G be a Gröbner basis for I w.r.t. $\underset{A}{>}$. The set N_I admits a 0-standard cone decomposition Q, which may be found using the algorithm SPLIT$(1, X, \text{Hterm}_A(G))$. Let $d = 1 + \max\{\deg(h) : \langle h, u\rangle \in Q\}$. Theorem 4.11 assures that $\{g \in G : \deg(g) \leq d\}$ is also a Gröbner basis for I. The construction using algorithm EXACT then shows that the unique Macaulay constant b_0 is $\geq d$, and hence is also a bound on the degree of polynomials required in the Gröbner basis. $\square$

8. A bound for Gröbner basis degree. Let $F = \{f_1, \cdots, f_r\}$ be a homogeneous basis for an ideal I. Assume without loss of generality that f_1 has the largest degree $\deg(f_1) = d$. In the previous section, it has been shown that for any ideal I, there exists an exact partition Q for N_I in which the constant $\overline{a}_Q$ is zero. Furthermore, if the Macaulay constants associated with Q are $b_0 \geq b_1 \geq \cdots \geq b_{n+1} = 0$, then for degrees $z \geq b_0$, the Hilbert function of N_I attains the polynomial form

$$\varphi_{N_I}(z) = \binom{z+n}{n} - 1 - \sum_{i=1}^{n} \binom{z - b_i + i - 1}{i} .$$

It also has been shown that I itself has a direct decomposition consisting of the principal ideal (f_1) and an exact partition P with $\overline{a}_P = d$. Let $a_0 \geq a_1 \geq \cdots \geq a_{n+1} = d$ be the Macaulay constants for the portion of I partitioned by P. Then, for degrees $z \geq a_0$ the Hilbert function of I is equal to the polynomial

$$\varphi_I(z) = \binom{z - d + n - 1}{n-1} + \binom{z - d + n}{n} - 1 - \sum_{i=1}^{n} \binom{z - a_i + i - 1}{i} .$$

Now since I and N_I form a direct decomposition of $K[X]$, the sum of their Hilbert functions must be equal to the Hilbert function of $K[X]$, which is

$$\varphi_{K[X]}(z) = \binom{z+n-1}{n-1} .$$

Therefore, for $z \geq \max\{a_0, b_0\}$,

$$\begin{aligned}\binom{z+n-1}{n-1} &= \binom{z-d+n-1}{n-1} + \binom{z-d+n}{n} \\ &\quad + \binom{z+n}{n} - 2 \\ &\quad - \sum_{i=1}^{n} \left[\binom{z-a_i+i-1}{i} + \binom{z-b_i+i-1}{i} \right] . \end{aligned} \tag{1}$$

The backwards difference operator ∇ is defined for any function $F(z)$ by $\nabla F(z) = F(z) - F(z-1)$, and $\nabla^j F(z) = \nabla(\nabla^{j-1} F(z))$. Using the identity

$$\binom{z+k}{n} - \binom{(z-1)+k}{n} = \binom{z+k-1}{n-1}$$

we have

$$\nabla \binom{z+k}{n} = \binom{z+k-1}{n-1} .$$

It then follows inductively that

$$\nabla^j \binom{z+k}{n} = \binom{z+k-j}{n-j} .$$

If $F_1(z) = F_2(z)$ for $z > k$, then clearly $\nabla F_1(z) = \nabla F_2(z)$ for $z > k+1$. For each j in the range $j = 0, \cdots, n-1$, apply the operator ∇^j to (1). [1] This yields the following set of equations for $j = 1, \cdots, n-1$, which are valid for large enough z:

$$\begin{aligned}\binom{z+n-j-1}{n-j-1} &= \binom{z-d+n-j-1}{n-j-1} + \binom{z-d+n-j}{n-j} \\ &\quad + \binom{z+n}{n} - 2 \\ &\quad - \sum_{i=j+1}^{n} \left[\binom{z-a_i+i-j-1}{i-j} + \binom{z-b_i+i-j-1}{i-j} \right] . \end{aligned}$$

Each side of these equations is a polynomial in z, so they must agree for each power of z. In particular, they must have the same constant term. Note that the constant term of $\binom{z+k}{n}$ is given by

$$\binom{0+k}{n} = \begin{cases} \binom{k}{n}, & k \geq 0, \\ (-1)^n \binom{n-1-k}{n}, & k < 0 . \end{cases}$$

Taking the constant terms of the previous set of equations, we obtain

$$\begin{aligned} 1 &= (-1)^{n-j-1} \binom{d-1}{n-j-1} + (-1)^{n-j} \binom{d-1}{n-j} \\ &\quad -1 - \sum_{i=j+1}^{n} (-1)^{i-j} \left[\binom{a_i}{i-j} + \binom{b_i}{i-j} \right] . \end{aligned}$$

[1] The technique of using the backwards difference operator has been used in a slightly different manner in [10].

At $j = n-1$, this is simply

$$1 - \binom{d-1}{1} - 1 + a_n + b_n = 1 .$$

So $a_n + b_n = d$. Together with the conditions $a_n \geq d$ and $b_n \geq 0$, this implies that $a_n = d$. When we substitute these values, the series of equations becomes

$$2(-1)^{n-j-1}\binom{d-1}{n-j-1} - 1 - \sum_{i=j+1}^{n-1} (-1)^{i-j}\left[\binom{a_i}{i-j} + \binom{b_i}{i-j}\right] = 1 .$$

Let c_{j+1} denote the sum $a_{j+1} + b_{j+1}$. Solving for this expression yields

$$c_{j+1} = 2 + 2(-1)^{n-j}\binom{d-1}{n-j-1} + \sum_{i=j+2}^{n-1} (-1)^{i-j}\left[\binom{a_i}{i-j} + \binom{b_i}{i-j}\right] .$$

At this point, we may note that the sum on the right is vacuous for $j = n-2$ and conclude that $c_{n-1} = 2 + 2(d-1) = 2d$. And since

$$\binom{a_{i+1}}{k} + \binom{b_{i+1}}{k} \leq \binom{c_{i+1}}{k} \tag{2}$$

is true for all i, for $j = n-3$ we have

$$c_{n-2} \leq 2 - 2\binom{d-1}{2} + \binom{2d}{2} = d^2 + 2d .$$

The remaining equations ($j < n-3$), all contain the expression

$$2 + (-1)^{n-j}\left[2\binom{d-1}{n-j-1} - \binom{a_{n-1}}{n-j-1} - \binom{b_{n-1}}{n-j-1}\right] .$$

The magnitude of this combination is bounded by

$$\binom{c_{n-1}}{n-j-1} ,$$

so the inequalities above may be replaced with the weaker inequalities:

$$c_{j+1} \leq \binom{c_{n-1}}{n-j-1} + \sum_{i=j+2}^{n-2} (-1)^{i-j}\left[\binom{a_i}{i-j} + \binom{b_i}{i-j}\right] .$$

The term in the sum for $i = j+3$ has a negative sign, and hence this term may be discarded. Giving all the remaining terms a positive sign produces the following still weaker inequalities :

$$\begin{aligned} c_{j+1} &\leq \binom{c_{n-1}}{n-j-1} + \left[\binom{a_{j+2}}{2} + \binom{b_{j+2}}{2}\right] + \sum_{i=j+4}^{n-2}\left[\binom{a_i}{i-j} + \binom{b_i}{i-j}\right] \\ &\leq \binom{c_{j+2}}{2} + \sum_{i=j+4}^{n-1}\binom{c_i}{i-j} . \end{aligned}$$

Or, upon repairing the subscripts by the change $j \to j-1$:

$$c_j \leq \binom{c_{j+1}}{2} + \sum_{i=j+3}^{n-1} \binom{c_i}{i-j+1}.$$

These inequalities may now be solved inductively to provide a bound on the magnitude of c_j.

LEMMA 8.1. *For $j \leq n-2$, the value of c_j satisfies the inequality $c_j \leq D_j$, where*

$$D_j = 2\left(\frac{d^2}{2}+d\right)^{2^{n-j-1}}.$$

Proof. It was already determined that $c_{n-2} \leq d^2+2d$, satisfying this claim. Now, assume inductively that c_i has the indicated bound for $j < i \leq n-2$.

For $i \geq j+3$ the inequality $2^{i-j-1} \geq i-j+1$ can be used to see that $(2^{n-i-1})(i-j+1) \leq 2^{n-j-2}$. Therefore,

$$\binom{D_i}{i-j+1} \leq \frac{D_i^{i-j+1}}{(i-j+1)!} \leq \frac{D_i^{2^{i-j-1}}}{(i-j+1)!} = D_{j+1}\frac{2^{i-j}}{(i-j+1)!}.$$

And so,

$$\begin{aligned} c_j &\leq \binom{c_{j+1}}{2} + \sum_{i=j+3}^{n-1} \binom{c_i}{i-j+1} \\ &\leq \binom{D_{j+1}}{2} + \sum_{i=j+3}^{n-1} \binom{D_i}{i-j+1} \\ &\leq \frac{D_{j+1}^2 - D_{j+1}}{2} + \sum_{i=j+3}^{n-1} D_{j+1}\frac{2^{i-j}}{(i-j+1)!} \\ &\leq \frac{D_{j+1}^2}{2} - D_{j+1}\left[\frac{1}{2} - \sum_{i=j+3}^{n-1}\frac{2^{i-j}}{(i-j+1)!}\right] \\ &\leq \frac{D_{j+1}^2}{2} = D_j. \end{aligned}$$

□

From this, we may conclude that the Macaulay constants a_1 and b_1 are each less than $D_1 = 2((d^2/2)+d)^{2^{n-2}}$. But what about the constants a_0 and b_0 that did not appear explicitly in the Hilbert function? For z in the range $\max\{a_1,b_1\} < z \leq \max\{a_0,b_0\}$, use the equality

$$\varphi_I(z) + \varphi_{N_I}(z) = \varphi_{K[X]}(z)$$

to obtain the relation

$$\begin{aligned} &(\overline{\varphi}_I(z) + |\{\langle h,\emptyset\rangle \in P \,:\, \deg(h) = z\}|) \\ &\quad +(\overline{\varphi}_{L_I}(z) + |\{\langle h,\emptyset\rangle \in Q \,:\, \deg(h) = z\}|) \;=\; \varphi_{K[X]}(z). \end{aligned}$$

At this point, we may note that the relationship $\overline{\varphi}_I(z) + \overline{\varphi}_{L_I}(z) = \varphi_{K[X]}(z)$, which was claimed valid for $z > \max\{a_0, b_0\}$ actually holds for $z \geq \max\{a_1, b_1\}$. Therefore, for z in the range $\max\{a_1, b_1\} < z \leq \max\{a_0, b_0\}$, it must be the case that

$$(|\{\langle h, \emptyset\rangle \in P \,:\, \deg(h) = z\}|) + (|\{\langle h, \emptyset\rangle \in Q \,:\, \deg(h) = z\}|) \;=\; 0.$$

This implies that $P \cup Q$ contains no pair $\langle h, \emptyset\rangle$ with $\deg(h) > \max\{a_1, b_1\}$. Therefore, the constant D_1 is also a bound on the value of b_0. Using this bound within Lemma 7.2 provides the proof of the following theorem.

THEOREM 8.2. *Let I be an ideal of $K[X] = K[x_1, \cdots, x_n]$ generated by a set of homogeneous polynomials F. Let $d = \max\{\deg(f) : f \in F\}$. Then for any admissible ordering $>_A$, the degree of polynomials required in a Gröbner basis for I with respect to $>_A$ is bounded by $2((d^2/2) + d)^{2^{n-2}}$.*

For I an affine ideal, we can homogenize a basis F for I using one additional variable x_{n+1}. Therefore, for any set of polynomials F we have Corollary 8.3.

COROLLARY 8.3. *Let $F \subset K[X]$, I the ideal generated by F, and let d be the maximum degree of any $f \in F$. Then for any admissible ordering $>_A$, the degree of polynomials required in a Gröbner basis for I with respect to $>_A$ is bounded by $2((d^2/2) + d)^{2^{n-1}}$.*

REFERENCES

[1] D. BAYER, *The division algorithm and the Hilbert scheme*, Ph.D. thesis, Harvard University, Cambridge, MA, 1982.

[2] B. BUCHBERGER, *A criterion for detecting unnecessary reductions in the construction of Gröbner-basis*, in Lecture Notes in Computer Science, Vol. 72, Springer-Verlag, Berlin, New York, 1979, pp. 3–21.

[3] ———, *Gröbner basis: An algorithmic method in polynomial ideal theory*, in Multidimensional Systems Theory, N. K. Bose, ed., D. Reidel, Boston, MA, 1985, pp. 184–229.

[4] ———, *History and basic features of the critical-pair/completion procedure*, J. Symb. Comput., 3 (1987), pp. 3–38.

[5] T. DUBÉ, *Quantitative analysis of problems in computer algebra: Gröbner bases and the Nullstellensatz*, Ph.D. thesis, New York University, New York, 1989.

[6] M. GIUSTI, *Some effectivity problems in polynomial ideal theory*, in Lecture Notes in Computer Science, Vol. 174, Springer-Verlag, Berlin, New York, 1984, pp. 159–171.

[7] F. S. MACAULAY, *Some properties of enumeration in the theory of modular systems*, Proc. London Math. Soc., 26 (1927), pp. 531–555.

[8] E. W. MAYR AND A. R. MEYER, *The complexity of the word problems for commutative semigroups and polynomial ideals*, Adv. in Math., 46 (1982), pp. 305–329.

[9] B. MISHRA AND C. K. YAP, *Notes on Gröbner basis*, in Information Sciences, An International Journal, Vol. 48, Elsevier Science, New York, 1989, pp. 219–252.

[10] M. MÖLLER AND F. MORA, *Upper and lower bounds for the degree of Groebner bases*, in Lecture Notes in Computer Science, Vol. 174, Springer-Verlag, Berlin, New York, 1984, pp. 172–183.

[11] R. P. STANLEY, *Hilbert functions of graded algebras*, Adv. in Math., 18 (1978), pp. 57–83.

[12] O. ZARISKI AND P. SAMUEL, *Commutative Algebra*, Vol. 2, Springer-Verlag, Berlin, New York, 1960.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 775-786, October 1990

THE SPATIAL COMPLEXITY OF OBLIVIOUS k-PROBE HASH FUNCTIONS*

JEANETTE P. SCHMIDT† AND ALAN SIEGEL‡

Abstract. The problem of constructing a dense static hash-based lookup table T for a set of n elements belonging to a universe $U = \{0, 1, 2, \cdots, m-1\}$ is considered. Nearly tight bounds on the spatial complexity of oblivious $O(1)$-probe hash functions, which are defined to depend solely on their search key argument, are provided. This establishes a significant gap between oblivious and nonoblivious search. In particular, the results include the following:

- A lower bound showing that oblivious k-probe hash functions require a program size of $\Omega((n/k^2)e^{-k} + \log\log m)$ bits, on average.
- A probabilistic construction of a family of oblivious k-probe hash functions that can be specified in $O(n\,e^{-k} + \log\log m)$ bits, which nearly matches the above lower bound.
- A variation of an explicit $O(1)$ time 1-probe (perfect) hash function family that can be specified in $O(n + \log\log m)$ bits, which is tight to within a constant factor of the lower bound.

Key words. hashing, oblivious, spatial complexity, power bound, perfect hashing

AMS(MOS) subject classifications. 68P05, 68P10, 68Q05, 68R05, 68R10

1. Introduction.

Hashing is one of the most important and commonly used methods to organize simple collections of information (see References for a partial list). The applications are extensive, and the subject has a correspondingly rich theoretical literature. In this paper, we will be restricting our attention principally to the dictionary problem, which concerns how to organize a set S of distinct keys within a table T so that the elements can be retrieved quickly. We shall take the data set to be static, so that the hash table need not support insertion or deletion, and consider only open addressing, so that pointers will not be used. Most of our results will focus on 100 percent utilized tables.

The history of this problem, which we detail in the next subsection, would seem to suggest that virtually all questions have been answered for this specific problem; a variety of lower bounds have been established [Y81] and [Me84], and elegant constructions have been discovered, which nearly meet the lower bounds [FKS84].

Recently, however, new techniques for organizing data have been devised [FMNSSS88], [FNSS88], which show that an enormous amount of information can be encoded within a search table. The thrust of these results is to show how to exploit nonoblivious search, which can use an adaptive probe strategy based upon information gleaned from unsuccessful probes. The consequences are performance bounds and extensions for the dictionary problem that had been believed to be impossible, for $O(1)$-probe hashing schemes [FNSS88].

* Received by the editors January 18, 1989; accepted for publication (in revised form) October 11, 1989.

† Computer Science Department, Polytechnic University, 333 Jay Street, Brooklyn, New York 11201. The work of this author was supported in part by Defense Advanced Research Projects Agency grant F49620-87-C-0065, and was in part carried out while the author was visiting the Computer Science Department at Rutgers University.

‡ Computer Science Department, Courant Institute, New York University, New York, New York 10012. The work of this author was supported in part by National Science Foundation grants CCR-8902221 and CCR-8906949 and Office of Naval Research grant N00014-85-K-0046. A preliminary version of this work was published in §§ 1-3 of "On aspects of universality and performance for closed hashing," [SS89].

The unexpected opportunities demonstrated by these results have led us to examine afresh the computational models and assumptions underlying the lower bound of [Me84] and the construction of [FKS84]. These results are for 1-probe schemes, which by definition cannot be adaptive. The natural question to ask is whether the opportunity to use the information encoded within the constant number of probed locations is genuinely significant, or if the power of the [FNSS88] scheme is actually due to the ability to query several locations for the search key.

A few preliminary definitions help formalize the problem. We let the data set S comprise n elements belonging to the universe $U = \{0, 1, 2, \cdots, m-1\}$. Our hash-based lookup table T is also of size n. A sequence $H = (h_1, h_2, \cdots, h_k)$ of functions is a k-probe hash function for S, if $H: U \mapsto [1, n]^k$, and $T[1 \cdots n]$ can be organized so that each item $s \in S$ is located is one of the k probe positions defined by applying the k-probe functions to s.

The method of probing must be defined quite precisely. A hashing strategy is **oblivious** if the search locations computed by H are based solely on the key s, and not on other keys stored in T. In this case, the search strategy is

```
for i ← 1, 2, ···, k do
    if T[h_i(s)] = s then return (h_i(s))
endfor;
if s has not been found then return (FAIL).
```

In contrast, a **nonoblivious** hashing scheme can make computational use of the keys encountered during unsuccessful search, which offers a wider and conceivably more powerful range of search strategies. Formally, such a hashing scheme differs in that h_i is an i-ary function mapping U^i into $[1, n]$. The nonoblivious search strategy is

```
for i ← 1, 2, ···, k do
    s_i ← T[h_i(s, s_1, s_2, ···, s_{i-1})];
    if s_i = s then return (h_i(s, s_1, s_2, ··· s_{i-1}))
endfor;
if s has not been found then return (FAIL).
```

A family $\bar{H}$ is a *k-probe* family for U if every n-element subset $S \subset U$ has some k-probe hash function in $\bar{H}$. A 1-probe hash function for S is called a *perfect* hash function, and a perfect family for U denotes a corresponding 1-probe family. (In the exposition that follows, we shall frequently take the liberty of suppressing the implicitly understood parameters n, m, and even k, for notational convenience.) The principal problem we analyze is how many hash functions are needed to define a *k-probe* family $\bar{H}$, in the oblivious hashing model. In particular, we attain estimates for $\log(\bar{H})$, which is the number of bits needed to specify a particular hash function $H \in \bar{H}$.

1.1. Background. Mehlhorn showed that the bit length of a perfect hash function mapping S into a table T of size n_1, where $n \leqq n_1 < (1-\varepsilon)m$, is lower bounded by $\Omega(n^2/n_1 + \log\log m - \log\log n_1)$ [Me82], [Me84]. The explicit construction of efficient perfect hash functions has been explored, among others, by [Me84], [Ma83], [FKS84], [JvE86], and [SvE84]. In [FKS84], an $O(1)$-time computable perfect hashing scheme for full tables (that is to say, $n_1 = n$) is presented, which requires a description of $O(\log\log m + n\sqrt{\log n})$ bits. Jacobs and van Emde Boas [JvE86] reduce the upper bound for $O(1)$ time 1-probe (FKS-like) hashing to $O(\log\log m + n\log\log n)$-bits. Slot and van Emde Boas [SvE84] show that a variation of the FKS-scheme can be made space optimal at a cost of taking $O(n)$ time to hash a value. We show that a space-optimal variation can be attained while maintaining the $O(1)$ time performance for hashing.

Nonoblivious $O(1)$-probe schemes turn out to be much more powerful than 1-probe schemes, where the question of obliviousness, of course, has no bearing. A nonoblivious scheme that needs only $O(\log\log m+\log n)$ bits is presented in [FNSS88] and [FN88] has recently shown that no additional memory is required when m is polynomial in n. No comparable oblivious schemes are known, and the natural question to resolve is how much of the exponential spatial advantage of $O(1)$-probe nonoblivious schemes over 1-probe schemes is due to their adaptive character, and how much is due to the opportunity to perform additional oblivious search. The counting arguments used in [Me84] seem to provide no help in an effort to construct lower bounds for multiprobe oblivious hashing: the $\Omega(n)$-bit portion of the argument collapses even for 3-probe schemes.

We show that the spatial complexity of k-probe oblivious hash functions for full tables is lower bounded by $\Omega(n\alpha^k+\log\log m)$, and that this bound is tight with $e^{-1-2\log k/k}\leqq\alpha\leqq e^{-1+5/k}$. It is worth noting that in contrast to 1-probe schemes, no comparable lower bound can be obtained for $O(1)$-probe schemes for load factors less than 1. Indeed, a probabilistic construction shows that k-probe oblivious hash functions can be specified in as few as $O(\log n+\log\log m)$ bits, when the size n_1 of the hash table is $(1+\varepsilon)n$, $\varepsilon>0$.

For completeness, we note that Mairson [Ma83], [Ma84] analyzed a number of related problems, including binary search adapted to a page oriented hash scheme, where the cost to read a 2^k-record page is fixed, and the data is sorted on each page. He analyzed a scheme limited to reading one page and supporting k rounds of binary search. While his scheme is not oblivious since it uses binary search, its spatial complexity is remarkably close to the spatial complexity of fully obvious k-probe schemes, analyzed in this manuscript.

In § 2, we prove that the spatial complexity of a k-probe oblivious hash function for a set of n elements from the universe $U=[1, m]$ is $\Omega(e^{-k}n/k^2+\log\log m)$. Section 3 shows that, with probability $(1-o(1))$, a random set of $(2^{O(ne^{-k})}\log m)$ k-probe hash functions contains a perfect k-probe hash function for every n-element subset of U. Section 4 exhibits an explicit $O(1)$-time 1-probe hash scheme that uses only $O(\log\log m+n)$ bits of external memory, and thereby shows that the lower bound for 1-probe schemes can be met for $O(1)$ time hash functions.

2. Lower bounds for oblivious search. Mehlhorn's $\Omega(n)$ bound for the size of a perfect hash function [Me82], is based on the following counting argument: the number of functions must be at least the number of n-item subsets, which belong to an m-element universe U, divided by the maximum number of subsets that can be mapped one-to-one into $[0, n-1]$ by a single hash function: $count\geqq\binom{m}{n}/(m/n)^n$. Taking the logarithm of this estimate gives the size bound for such hash functions. Unfortunately, this ratio decreases by a factor of k^n when k-probe maps are allowed. Formally, a hash function defines a bipartite graph on $B=U\times\{0,\cdots,n-1\}$. In the 1-probe case, each vertex in U has degree 1, and the count of the number of possible coverings of $\{0,\cdots,n-1\}$, (subgraphs in which each vertex in $\{0,\cdots,n-1\}$ has degree 1), afforded by a single function is precisely the number of different subsets serviceable by the hash function. Once k probes are allowed, the degree of each vertex increases by a factor of k. When the resulting bipartite graph supports a perfect matching for some n-element subset A in U, this perfect matching can be used to store A in the table and the k-probe hash function can retrieve it. The number of coverings of $\{0,\cdots,n-1\}$ is easily estimated (by $(km/n)^n$), but may overcount the number of serviceable subsets by as much as a factor of k^n, for $k>1$.

Consequently, we are obliged to model the k-probe scheme more carefully. We model a family of such hash functions as a set $\bar{H}$ of bipartite graphs on B, where each node of U has degree k, for each graph. $\bar{H}$ contains a k-probe perfect hash function for each n-element subset of U if and only if each n-element subset has a perfect matching for some $H \in \bar{H}$. It should be emphasized that these performance bounds are for oblivious probe schemes with sufficient probing to store the data: given a suitable bipartite graph, the data cannot be successfully stored without some means of extracting a matching within the graph. Algorithms for such problems are well known [HK73], and probabilistic incremental approaches have also been analyzed in connection with hashing [SS80]. In any case, the lower bound for the number of bits needed to describe such a k-probe oblivious hash function is thus $\log |\bar{H}|$.

To estimate $|\bar{H}|$, we choose an arbitrary $H \in \bar{H}$ and a randomly selected n-element set $V \subset U$ and compute an upper bound $\bar{p}$ for the probability that $H \in \bar{H}$ provides a perfect matching for V. It then follows that the number of n element subsets of U, for which h is perfect is at most $\binom{m}{n}\bar{p}$. Thus

$$|\bar{H}| \geqq 1/\bar{p}$$

and members of $\bar{H}$ cannot be identified in fewer than $\log_2 (1/\bar{p})$ bits.

By letting V be selected at random, we may imagine an honest intermediary who conveys, as answers to queries by our (partial) matching algorithm, precise (minimal amounts of) information about the items selected.

We will estimate the probability that a carefully selected subset $\tilde{\eta} \in [1, n]$ of n/ck items is covered by edges emanating from V. (The exact value of c will be specified later.)

For $v \in U$, we define $Image(v) = \{i \in [1, n] | \{v, i\}$ is an edge in the graph $H\}$,

For $i \in [1, n]$ we define $Preimage(i) = \{v \in U | \{v, i\}$ is an edge in the graph $H\}$.

Let $P_i = |Preimage(i)|$.

Let V_i be an ordered set of elements denoted by $(v_1^i, v_2^i, \cdots)$.

The selection process for $\tilde{\eta}$, which is based on the fact that elements with small P_i value in η are not overly likely to be covered by the edges emanating from a randomly selected set V, proceeds as follows:

0. Let $V_0 \leftarrow V$, $\eta_0 \leftarrow \{1, \cdots, n\}$, $\tilde{\eta} \leftarrow \emptyset$.
1. For $i \leftarrow 0$ to $n/ck - 1$ perform 2 through 5:
2. Let ν_i be an element in η_i with minimum P_{ν_i}, and set $\tilde{\eta} \leftarrow \tilde{\eta} \cup \{\nu_i\}$.
3. Sequence through the elements of V_i and stop at the first $v^i \in V_i$ that is in $Preimage(\nu_i)$. If no such v^i is found return "fail", *H is not perfect for V.*
4. $V_{i+1} \leftarrow V_i - \{v^i\}$.
5. Set $\eta_{i+1} \leftarrow \eta_i - Image(v^i)$.
   Note that all the preimages of items in η_{i+1} are contained in V_{i+1}, and $|\eta_{i+1}| \geqq n - (i+1)k \geqq n - n/c$.

If the above procedure fails, then there is no matching from V to $[1, n]$. (The converse is not true, of course, but our aim is to upper bound the probability of a success.)

We have to estimate the probability that V contains an element in $Preimage(\nu_i)$, for $0 \leqq i < n/ck$, at step 3. The query at step 3 is whether $v^i \in V_i$ is in $Preimage(\nu_i)$. Only at step 5 is $Image(v^i)$ actually revealed. Since $P_{\nu_0} \leqq km/n$, the conditional probability that v_j^0, the jth element in V_0, covers ν_0 given that the first $j-1$ do not, is

$$\text{Prob}\{v_j^0 \in Preimage(\nu_0) | v_1^0, \cdots, v_{j-1}^v \notin Preimage(\nu_0)\} \leqq \frac{km}{n(m-(j-1))} \leqq \frac{k}{n(1-n/m)}.$$

The estimates of the probabilities of success for $i > 0$, in step 3, are slightly more delicate. In particular, the event: "$v_j^i \in Preimage(\nu_i) | v_1^i, \cdots, v_{j-1}^i \notin Preimage(\nu_i)$" has

to be conditioned by the additional information concerning v_j^i, acquired by previous queries. However, all we have learned is that v_j^i is not in the *Preimage* of some of the elements $\nu_0, \nu_1, \cdots, \nu_{i-1}$, that $v_j^i \notin V - V_i$ and that $v_j^i \notin \{v_1^i, \cdots, v_{j-1}^i\}$. The probability that $v_j^i \in$ *Preimage*(ν_i) is maximized if *none* of the elimimated candidates are in *Preimage*(ν_i). Since *Preimage*(ν_l) (where ν_l is selected in step 2) is among the $kl+1$ smallest *Preimages* of elements in η_0, it follows that $\sum_{l=0}^{i-1} P_{\nu_l} \leqq ikm/n$; also $V - V_i \subset \bigcup_{l=0}^{i-1}$ *Preimage*(ν_l), and therefore in the most extreme case, the conditional information can eliminate at most $(\sum_{l=0}^{i-1} P_{\nu_i} + j - 1 < m/c + n)$ elements of U as candidates for v_j^i. It follows that

$$\text{Prob}\{v_j^i \in \textit{Preimage}(\nu_i) \,|\, \text{all previous events}\} \leqq \frac{P_{\nu_i}}{(m - m/c - n)},$$

and

$$\begin{aligned}\text{Prob}\{\nu_i \in \textit{Image}(V_i) \,|\, \text{all previous events}\} &\leqq 1 - \left(1 - \frac{P_{\nu_i}}{m - m/c - n}\right)^{n-i} \\ &\leqq 1 - \left(1 - \frac{P_{\nu_l}}{m - m/c - n}\right)^{n}.\end{aligned}$$

And finally

$$\begin{aligned} p &= \text{Prob}\{H \text{ is perfect for } V\} \\ &\leqq \text{Prob}\{\text{that the } \textit{Image} \text{ of } V \text{ contains all the elements in } \tilde{\eta}\} \\ &\leqq \prod_{i=0}^{n/ck-1} \left(1 - \left(1 - \frac{P_{\nu_i}}{m - m/c - n}\right)^{n}\right).\end{aligned}$$

Since $\sum_{l=0}^{i-1} P_{\nu_l} \leqq ikm/n$, by construction, the above product is maximized when all P_{ν_l} are set to km/n. Hence

$$\begin{aligned} p = \text{Prob}\{H \text{ is perfect for } V\} &\leqq \left(1 - \left(1 - \frac{km/n}{m - m/c - n}\right)^{n}\right)^{n/ck} \\ &\leqq \left(1 - (1 - o(1)) \exp\left(-\frac{km}{m - m/c - n}\right)\right)^{n/ck}.\end{aligned}$$

We choose $c = (k+1)m/(m-n)$ and conclude that

$$p \leqq \bar{p} = (1 - (1 - o(1))\, e^{-(k+1)/(1-n/m)})^{n(1-n/m)/(k+1)k}.$$

Computing $\log_2(1/\bar{p})$ gives roughly $((n/k^2)\, e^{-(k+1)/(1-n/m)})$ bits necessary to specify an oblivious search strategy. We have proved Theorem 1.

THEOREM 1. *The spatial complexity of a k-probe oblivious hash function for a set of n elements belonging to the universe* $[1, m]$ *is* $\Omega((n/k^2)\, e^{-k(m/(m-n))})$ (or $\Omega((n/k^2)\, e^{-k})$ *for* $n = o(m)$).

This lower bound for $|\bar{H}|$, as we shall see, captures the complexity resulting from the size parameter n, but it is quite insensitive to the growth of m. A simple information theoretic argument, however, which was mentioned to us by Fiat and Naor and also

independently by Fredman, shows that $\Omega(\log m/(k \log n))$ is also a lower bound on the size of $|\bar{H}|$. The argument is given here for completeness. We may encode the target sets, under the $|\bar{H}|$ k-probe hash functions, of the elements in U as one function g from U to $\binom{n}{k}^{|\bar{H}|}$. One of the conditions a k-probe scheme satisfies is that no $k+1$ items in U have all k probe locations in common, and thus no $k+1$ items may have the same image under g. It follows that $m \leqq k\binom{n}{k}^{|\bar{H}|}$, and thus

$$|\bar{H}| \geqq \frac{\log (m/k)}{\log \binom{n}{k}},$$

and $\Omega(\log \log (m) - k \log (n))$ bits are needed to name H.

Combining the information theoretic bound with Theorem 1 we get Theorem 2.

THEOREM 2. *The spatial complexity of a k-probe oblivious hash function for a set of n elements belonging to the universe* $[1, m]$ *is* $\Omega(\log \log m + (n/k^2)\, e^{-k(m/(m-n))})$.

By appealing to a hypergraph model and extending the proof technique of Theorem 1, the method of Fredman and Komlós [FK84] can be incorporated to show that $|\bar{H}|$ is actually lower bounded by the product of the two bounds, rather than the maximum of the two, which gives a slightly stronger result.

Theorem 2 also shows that the $\Omega(n)$ portion of the lower bound for the spatial complexity of oblivious hash functions holds even for relatively small universes (such as $m \approx 2n$). On the other hand, for $k > 1$, our lower bound proof collapses if we were to map n elements into a table size $(1+\varepsilon)n$. The upper bound in Observation 5 of § 3 shows that the lower bound indeed does not hold for $\varepsilon > 0$.

3. Upper bounds for oblivious search.

3.1. Oblivious *k*-probe schemes. Our lower bounds suggest that, while k-probe perfect hash schemes must have a reasonably large program length, in the case of oblivious search, the $k-1$ additional search opportunities might reduce the length by a factor of about e^k in its n dependence. We appeal to methods from the theory of random graphs to give a nonconstructive demonstration that this reduction is indeed possible, at least in a formal sense. The proof is actually a probabilistic construction of a k-probe oblivious hash function. For $k = 1$, such a probabilistic construction has been given in [Me82]. In fact, the lower and upper bound of [Me82] were both attained by essentially one argument. When additional probes are permitted, the lower bound argument, as we have seen, is more delicate, while the upper bound, as we now demonstrate, requires much more care.

The procedure is to imagine constructing a random bipartite graph G on $U \times [1, n]$, where each vertex in U has degree k. We then select a random set S of n items from U and estimate a lower bound for the probability p that G contains a perfect matching on S. With positive probability, a family $\bar{H}$ of $\log_{1/(1-p)} \binom{m}{n}$ such random graphs will, for each n-element subset in U, contain some graph that has a perfect matching for that set, and it therefore follows that such an $\bar{H}$ exists. The number of bits required to designate an element $H \in \bar{H}$ is then $\log \log_{1/(1-p)} \binom{m}{n}$.

Accordingly, let $S \subset U$, $|S| = n$ be the subset we wish to match. For convenience, let G be used to name the portion of our random graph that is restricted to $S \times [1, n]$. A few preliminary remarks and definitions will help simplify the subsequent exposition.

- Recall Hall's (a.k.a., the marriage) theorem: A matching exists if and only if for all j, the image of every set of j vertices in S contains at least j vertices (in $[1, n]$). Let Hall (j) be the property that the image of every set of j vertices in S contains at least j vertices.

Our aim is to lower bound the probability of the event (Hall $(k) \wedge \cdots \wedge$ Hall (n)). Our estimates for Hall (j) are based on *expected* number of sets of j vertices that

violate the Hall (j) property and hence are connected to fewer than j vertices:

$$\text{Hall}\,(j) \geqq 1 - \binom{n}{j}\binom{n}{j-1}\left(\frac{\binom{j-1}{k}}{\binom{n}{k}}\right)^{j}.$$

Unfortunately, the above estimates become useless for values of j close to n. For those cases we will estimate Hall (j), by running a matching algorithm on G and estimating the probability that the algorithm fails in a fairly late stage.

- Let, for each $s \in S$, the first three random edges that are chosen to emanate from s be gold. The remaining $k-3$ edges will be green.

Our analysis uses the following simple (not most efficient) matching algorithm, which is based on n successive breadth first searches from unmatched vertices. For each $s \in S$, a new breadth first search is initiated to find an augmenting path from s. A bfs level explores nonmatching edges from a current vertex set $R \subset S$, which is initially $\{s\}$. The search is successful for s if it discovers a vertex in $[1, n]$ that has not been visited by any of the previous searches, and is therefore unmatched. Otherwise, the currently matched mates (in S) of the $[1, n]$ vertices that are newly encountered in the current level of the search (i.e., that have not been previously seen by the current search) are used to replace R for the next level of exploration. Once a previously unmatched vertex $v \in [1, n]$ is visited, the alternating property of being matched or not, along the path of edges from s to v reversed. Then a new s is selected and a new search stage is initiated.

Our use of the algorithm has two constructive phases. The first phase pursues a partial matching on the subgraph of gold edges, until a set X of $n/4$ vertices of S have been matched, or a subset of $j \leqq n/4$ elements have been encountered that violates Hall (j). In the second phase, the count is extended, if possible, to n via both gold and green edges. If a match does not occur by the time $(k-4)n$ green edges have been looked at (in phase 2), the algorithm aborts and fails. The postamble can be entered upon failure encountered during phases 1 or 2, or upon a successful matching (completed in phase 2). If success or failure occurs before $(k-4)n$ green edges are explored, we may imagine that the algorithm continues to select and report new green edges until the requisite number of edges is used.

- Let A be the event that the Hall (j) property holds for $j \leqq n/4$ for the subgraph restricted to the gold edges.
- Let B be the event that the Hall (j) property holds for $n/4 < j \leqq n - n/(k-3)$, where both gold and green edges are used.
- Our matching algorithm will be uncovering the random edges of G as the algorithm progresses. Let the set X comprise the first $n/4$ vertices of $[1, n]$ that are matched (initially by gold edges).
- Let C be the event that the *first* $(k-4)n$ green edges encountered by the algorithm cover $[1, n] - X$. We could imagine an adversary taking note of matched vertices, which will be announced whenever the partial matching is augmented. It will also be told of every green edge that is encountered by the matching algorithm.

Our algorithm can switch to the fail state in phase 1 only if A does not hold. In phase 2, failure can occur for any of four reasons:

(1) Hall (j) does not hold for $j \leqq n/4$ for the full (unrestricted) graph, in which case A does not hold.

(2) B does not hold.

(3) Hall (j) does not hold, for $j > n - n/(k-3)$, in which case the algorithm must have uncovered at least $j(k-3) \geqq n(k-4)$ green edges and hence C does not hold.

(4) No matching occurs before $n(k-4)$ green edges are used, which is to say that C does not hold.

These observations prove Lemma 3.

LEMMA 3. $\Pr\{\text{matching}\} \geqq \Pr\{A \cap B \cap C\}$.

Consequently, $\Pr\{\text{matching}\} \geqq \Pr\{B \cap C \mid A\} \Pr\{A\} \geqq (\Pr\{B \mid A\} + \Pr\{C \mid A\} - 1) \times \Pr\{A\}$.

We now estimate these probabilities. The edges, for each vertex, are selected without replacement (i.e., each k-tuple of edges comprises k distinct members). It is also helpful to denote $[1, n] - X$ by $\{y_1, y_2, \cdots, y_{3n/4}\}$. There follows

$$\Pr\{A\} \geqq 1 - \sum_{3 < j \leqq n/4} \binom{n}{j}\binom{n}{j-1}\left(\frac{j-1}{n}\frac{j-2}{n-1}\frac{j-3}{n-2}\right)^j$$

$$\geqq 1 - \sum_{3 < j \leqq n/4} \binom{n}{j}\binom{n}{j-1}\left(\frac{j-1}{n}\right)^{3j} = 1 - O(n^{-4}).$$

$$\Pr\{B \mid A\} \geqq 1 - \sum_{n/4 < j \leqq n(k-4)/(k-3)} \binom{n}{j}\binom{n}{j-1}\left(\frac{j-1}{n}\right)^{(k-3)j}$$

$$> 1 - o((4\text{-}5)^{3n/4}), \qquad k > 6.$$

Note that $\Pr\{C \mid A\} = \Pr\{C\}$. Let $hit(y)$ be the event that y is hit by one of the first $(k-4)n$ green edges examined.

$$\Pr\{C\} = \Pr\{hit(y_1)\} \times \Pr\{hit(y_2) \mid hit(y_1)\} \times \cdots \Pr\{hit(y_{3n/4}) \mid hit(y_1), \cdots, hit(y_{3n/4-1})\}$$

$$\geqq \left(1 - \left(\frac{n-1}{n}\right)^{n(k-5)}\right)^{3n/4} > (1 - e^{-k+5})^{3n/4}.$$

Since $1 - \Pr\{B \mid A\} = o(\Pr\{C \mid A\})$, for $k > 6$, and $\Pr\{A\} = 1 - o(1)$, the probability p of a matching is $\geqq (1 - e^{-k+5})^{3n/4}(1 - o(1))$.

As we have already observed, a family of $|\bar{H}| = \log_{1/(1-p)} \binom{m}{n}$ such random graphs can supply a hash function (graph) for all n-element subsets. Computing $\log_2(|\bar{H}|) \approx \log_2 \log_2 m + \frac{3}{4} e^{5-k} n$ gives an upper bound on the number of bits necessary to designate which random graph gives a k-probe perfect hash function for a given set S. If time and workspace are of no significance, we can follow the theme given for 1-probe hashing in [Me82], which is to use the lexicographically smallest collection $\bar{H}$ among all collections of such 2^b bipartite graphs that supports k-probe hashing for all n-item subsets of $[0, m-1]$. Then the b-bit number H names a graph within that collection (with respect to the lexicographic order within $\bar{H}$). The desired datum is found in at most k probes, as specified by the graph's edges.

We have shown Theorem 4.

THEOREM 4. *A k-probe hash function for a set of n elements belonging to the universe $[1, m]$ into a table of size n can be specified in $\log\log m + O(n e^{-k})$ bits, which is tight to within a factor of k^2 of the lower bound (or tight to within a constant factor for constant k).*

Finally, we remark that a straightforward application of Hall's theorem shows that if we were to map the elements of S into a table of size $(1+\varepsilon)n$, then the probability of a successful matching using k probes is $1 - o(n^{-(k+1)})$, for $k > \max\{3, 2(\ln(1+\varepsilon))^{-1}\}$. Observation 5 follows.

OBSERVATION 5. *A k-probe hash function for a set of n elements belonging to the*

universe $[1, m]$ *into a table of size* $(1+\varepsilon)n$, *with* $k > \max\{3, 2(\ln(1+\varepsilon))^{-1}\}$, *can be specified in* $O(\log\log m + \log n)$ *bits.*

4. Explicit construction of an optimal $O(1)$-time 1-probe scheme. The upper bound in the previous section uses a probabilistic construction to establish the spatial complexity of k-probe oblivious hash functions. Unfortunately, such a technique does not yield any explicit $O(1)$-time hash function. In this section we give a construction for an $O(1)$-time 1-probe scheme that uses auxiliary tables of only $O(n)$-bits and is therefore optimal in both time and space. The scheme is a variation of the perfect hashing scheme originating in [FKS84]; other variations appear in [SvE84] and [JvE86]. Interestingly, the construction has commonality with the $O(n)$ time variation presented in [SvE84].

The hashing technique given in [FKS84] uses a table of linear congruential functions of the form $(ax \bmod p) \bmod q$. The basic idea, which we detail below, is to use such a hash function to define an implicit partition of the data set into a favorably distributed collection of collision buckets. Given a bucket index, an explicit (locally defined) secondary hash function is looked up, which determines a unique address (within the bucket) for the item sought.

Altogether, the address evaluation uses a few $\log n$-bit arithmetic computations, several array accesses, and one long word computation, which consists of the modular reduction of a $\log m$ bit word and the subsequent (modular) multiplication of two $\log n + \log\log m$ bit words, and has the form $(kx \bmod p) \bmod n^2$, where $k, p \in U$ and $k < p < n^2 \log m$ and p is prime. It is natural to ask if the [FKS84] scheme or some other reasonable hash scheme can be expressed in an optimal $O(\log\log m + n)$ bits, while maintaining $O(1)$ time search, as conjectured in [SvE84].

We show that an [FKS84]-based scheme can be expressed in an optimal number of bits (up to a multiplicative factor) and can be performed with essentially the same selection of $O(1)$ arithmetic operations and array accesses.

We use the usual machine model associated with this problem, which is somewhat idealized. The model is basically that of a random access machine, although as in all other schemes which preceded this one, the standard RAM model is augmented with multiplicative arithmetic. In particular, an array access of an $O(\log n)$-bit word takes unit time, and index computations are permitted. Words can be added, subtracted, multiplied, and (integer) divided in constant time. We also use the same single long word computation as in [FKS84] and [Me82] to map elements from U into $[0, n^2]$. Similarly, we employ the expositional expediency of calling this computation an $O(1)$ time operation.

The original FKS construction has four basic steps.

(1) A (long word) function $h_\alpha(x)$ is found that maps S into $[0, n^2-1]$ without collisions, so that all subsequent computations can use normal length words. Corollary 2 and Lemma 2 in [FKS84] show that it is possible to set $h_\alpha(x) = (kx \bmod p) \bmod n^2$, with suitable $k < p < n^2 \log m$, where p is prime. Let Σ be the image of $h_\alpha \circ S$.

(2) Next, a function $h_\beta(x)$ is found that maps Σ into $[0, n-1]$ so that the sum of the squares of the collision sizes is not too large. Corollary 3 in [FKS84] shows that it is possible to set $h_\beta(x) = (\kappa x \bmod \rho) \bmod n$, where ρ is any prime greater than n^2 and $\kappa \in [0, \rho]$ so that $\sum_{0 \le j < n} |h_\beta^{-1}(j) \cap \Sigma|^2 < 3n$.

(3) For each hash bucket (i.e., integer having at least one appearance in the multiset $\{h_\beta(t)\}_{t \in \Sigma}$) a secondary hash function h_i is found that is one-to-one on the collision set. Let c_i be the size of the collision set for bucket i; $c_i = |h_\beta^{-1}(i) \cap \Sigma|$. Then for $x \in h_\beta^{-1}(i) \cap \Sigma$, $h_i(x) = (k_i x \bmod \rho) \bmod c_i^2$, where $k_i \in [0, \rho-1]$. The item $s \in S$, (which is represented by $t = h_\alpha(s)$ and located in hash bucket $i = h_\beta(t)$) is stored in

location $C_i + h_i(h_\alpha(s))$, where $C_i = c_0^2 + c_1^2 + \cdots + c_{i-1}^2$. This locates all n items within a size $3n$ table, say $A^*[1 \cdots 3n]$.

(4) Finally, the table is stored without vacant locations in an array $A[1 \cdots n]$, i.e., the array contains the items of S in the order of appearance in the table A^*.

The composite hash function requires the parameters k, p, κ, and ρ for h_α and h_β, a table $K[0 \cdots n]$ storing the parameters k_i for the secondary functions h_i, a table $C[0 \cdots n]$ listing the locations C_i, which separate elements from different buckets in A^*, and finally a compression table $D[1 \cdots 3n]$, where $D[j]$ gives the index, within A, of the item (if any) that hashes to the value j in A^* (by the function outlined in 1 through 3 above). As presented, the description of this perfect hash function needs $2 \log \log m + O(n \log n)$ bits. Fredman, Komlós, and Szemerédi [FKS84] then apply this same formulation for a rescaled number range (and a 2-level pair of tables for D) to achieve a hash function that is compressed to $2 \log \log m + O(n\sqrt{\log n})$bits.

Elementary information theoretic calculations can show that rescaling cannot be used to give a function size that matches the lower bound. Accordingly, we use the basic [FKS84] formulation given above. In choosing the secondary hash functions h_i for each bucket, we appeal to [FKS84]. They point out that since their existence proofs are based on expected case analysis, at least half of the numbers in $[0, \rho - 1]$ will yield appropriate functions h_i if the hash range is doubled. In particular, their Corollary 4 shows that given a collision set of any c_i items in Σ, at least half of the numbers in $[0, \rho - 1]$ will, if selected as multiplier k_i, yield a function $h_i(x) = (k_i x \bmod \rho) \bmod 2c_i^2$ that is one-to-one on the set. Consequently, if we have z collision buckets requiring multipliers, we may select a single multiplier that is one-to-one for $\lceil z/2 \rceil$ of the buckets, provided we double the size of the hash range in each case. Altogether, we need at most $\lfloor \log n \rfloor + 1$ different k_i values, where k_i is the ith multiplier servicing about $1/2^i$ of the buckets. In summary, the optimal perfect hash function is that described above, with two modifications:

(1) The (uncompressed) hash value is $h_i(x) = ((k_i x \bmod \rho) \bmod 2c_i^2) + 2C_i$.

(2) The multiplier values k are iteratively selected to satisfy the one-to-one requirement of the maximum number of (unsatisfied) h_i's.

Up to this point our method turns out to be essentially the same as the $O(n)$ time version presented in [SvE84].

Now the information content of the map from bucket indices into the $\lfloor \log n \rfloor + 1$ multipliers (i.e., the representation of the table K with Huffman coding) is $O(n)$. We must show how to achieve compact encodings of the tables K, C, and D that require only $O(n)$ bits, and readily are decodable in $O(1)$ time, in our computational model. The basic decoding operations we use are as follows:

(1) Extract a subsequence of bits from one word.

(2) Concatenate two bit strings of altogether $O(\log n)$ bits.

(3) Compute the bit-index of the kth zero in a word.

(4) Count the number of consecutive 1's in an $O(\log n)$-bit word, starting at the beginning.

(5) Count the number of zeros in an $O(\log n)$-bit word.

(6) Access a few constants.

The first two operations are a matter of arithmetic. The last four can be performed with small lookup tables. In particular, it suffices to break the words into words of $\log n/2$ bits (padded with leading zeros) and to use these words to access decoder arrays. The third operation, for example, requires for each k, $0 < k \leqq \log n/2$, an array *of* $\sqrt{n}$ words. Using operations 5 and 3 above, a word of $2 \log n$ (for example) bits requires up to four accesses to compute the location of a kth 0. The fourth and fifth operation are accomplished in a similar matter.

The table C, which contains the values $C_i(=\sum_{j=o}^{i-1} c_i^2)$, is encoded as follows. First, the values c_i^2 are stored in a table T_0 in unary notation (in order of appearance, separated by 0's). (Note that the bit length of T_0 is at most $4n$.) We let str_i denote the string that encodes c_i^2. Because of operations 1 and 2, we may suppose that each bit of T_0 is addressable. Let a_i be the address (index) of the starting point of $str_{i\lceil \log(4n)\rceil}$ in T_0, so that $0 \leqq a_i < 4n$, for $i = 1, \cdots, n/\lceil \log(4n)\rceil$. (Note that $a_i = C_{i\lceil \log(4n)\rceil} + i\lceil \log(4n)\rceil$.) A second table T_1, (of n bits), contains, in binary, the indices $a_1, \cdots, a_{n/\lceil \log(4n)\rceil}$, stored as $\lceil \log(4n)\rceil$ bit words. If $a_{i+1} - a_i < 2\lceil \log(4n)\rceil$, the table information for intermediate c_j^2 (i.e., for $i\lceil \log(4n)\rceil \leqq j < (i+1)\lceil \log(4n)\rceil$) can be readily decoded via $O(1)$ accesses to T_0 and T_1 and $O(1)$ of our decoding operations ((3) and (5)).

The case $a_{i+1} - a_i \geqq (\lceil \log(4n)\rceil)^2$ is handled via a third table T_2, (of size $4n$) which stores, starting in (bit) location a_i, $\lceil \log(4n)\rceil - 1$ binary indices for the starting locations of str_j, $i\lceil \log(4n)\rceil < j < (i+1)\lceil \log(4n)\rceil$ (in T_0).

The remaining case, $2\lceil \log(4n)\rceil \leqq a_{i+1} - a_i < (\lceil \log(4n)\rceil)^2$, is handled with an additional level of refinement. Let $b_{i,j}$ be the address (index in T_0) of the starting point of $str_{i\lceil \log(4n)\rceil + j\lceil \log\log(4n)\rceil}$, $(j < \lceil \log(4n)\rceil / \lceil \log\log(4n)\rceil$ and $a_i < b_{i,j} < a_{i+1})$. In T_2 we store, starting in location a_i, the binary offsets $b_{i,1} - a_i$, $b_{i,2} - a_i, \cdots$. Each offset is stored as a $2\lceil \log\log(4n)\rceil$-bit binary number. If $b_{i,j+1} - b_{i,j} \leqq 2 \log n$ the information for intermediate c_ℓ, $i\lceil \log(4n)\rceil + j\lceil \log\log(4n)\rceil < \ell < i\lceil \log(4n)\rceil + j\lceil \log\log(4n)\rceil$, is readily decoded (from table T_0); for all other cases, the offsets (of size $2\lceil \log\log(4n)\rceil$) of all intermediate c_ℓ's are encoded through one last level of indirection in a table T_3, starting at bit location $b_{i,j}$. (This last encoding requires $\lceil \log\log(4n)\rceil^2 \leqq \lceil \log n\rceil$.)

The table K can be encoded in exactly the same way. In particular, a table K_0 contains, for its ith sequence of bits, the integer α_i in unary, if k_{α_i} is the multiplier assigned to hash bucket i, $0 \leqq i < n$. (Recall that the first multiplier (encoded by the string "1") is usable for at least half of the hash buckets.) The total length of the sequence comprises at most n 0's, $n/2$ singleton 1's, $n/4$ doubleton 1's, etc., for a total length of $3n$. The multipliers $k_1, \cdots, k_{\log n}$ are stored in a $\log n$-word array.

It is evident that we have encoded a perfect hash function in $O(n)$ bits. To summarize, the above construction, combined with the lower bound of [Me84], gives Theorem 6.

THEOREM 6. *The spatial complexity of a $O(1)$-time perfect (1-probe) hash function for a set of n elements belonging to the universe $[1, m]$ is $\Theta(\log\log m + n)$.*

5. Conclusions and open problems. We have shown how to construct explicit space-time optimal perfect hash functions. It is worth noting that the construction illustrates the computational significance of a Random Access Machine model augmented with a size n^ε auxiliary program store.

In addition, we have given tight bounds on the spatial complexity of oblivious k-probe hash functions and have helped quantify the difference between oblivious and nonoblivious strategies. In particular, we have shown that, for full tables, $O(1)$ additional oblivious probes can reduce the requisite size of the search program by a constant factor, but no more. As is well known, the decrease in program size, for partially full search tables, is more dramatic: probabilistic arguments show the formal existence of oblivious $O(1)$ probe schemes that need only $O(\log\log m + \log n)$ bits.

It would be interesting to find an explicit construction of a small $O(1)$-probe oblivious family of hash functions that map n elements into a table of size $(1+\varepsilon)n$,[1] and to find such a family where function evaluation can be accomplished in $O(1)$ time.

[1] On recent results on the above problem, see [SS90] and [S89].

Acknowledgments. The authors thank A. Fiat, M. Fredman, M. Naor, J. Spencer, and P. van Emde Boas for helpful discussions.

REFERENCES

[A85] M. AJTAI, *A lower bound for finding predecessors in Yao's cell probe model*, IBM Tech. Report RJ 4867 (51297) Computer Science, October 3, 1985.

[AFK83] M. AJTAI, M. FREDMAN, AND J. KOMLÓS, *Hash functions for priority queues*, in Proc. 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, 1983, pp. 299-303.

[BBDOP86] F. BERMAN, M. E. BOCK, E. DITTERT, M. J. O'DONNELL, AND D. PLANK, *Collections of functions for perfect hashing*, SIAM J. Comput., 15 (1986), pp. 604-618.

[FMNSSS88] A. FIAT, I. MUNRO, M. NAOR, A. SCHÄFFER, J. P. SCHMIDT, AND A. SIEGEL, *An implicit data structure for searching a multikey table in logarithmic time*, J. Comput. System Sci., to appear.

[FN88] A. FIAT AND M. NAOR, *Implicit $O(1)$ probe search*, in Proc. 21st ACM Symposium on Theory of Computing, Seattle, WA, 1989, pp. 336-344.

[FNSS88] A. FIAT, M. NAOR, J. P. SCHMIDT, AND A. SIEGEL, *Non-oblivious hashing*, in Proc. 20th ACM Symposium on Theory of Computing, Chicago, IL, 1988, pp. 367-376.

[FK84] M. L. FREDMAN AND J. KOMLÓS, *On the size of separating systems and families of perfect hash functions*, SIAM J. Algebraic Discrete Methods, 5 (1984), pp. 61-68.

[FKS84] M. L. FREDMAN, J. KOMLÓS, AND E. SZEMERÉDI, *Storing a sparse table with $O(1)$ worst case access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538-544.

[G81] G. H. GONNET, *Expected length of the longest probe sequence in hash code searching*, J. Assoc. Comput. Mach., 28 (1981), pp. 289-304.

[GM79] G. H. GONNET AND J. I. MUNRO, *Efficient ordering of hash tables*, SIAM J. Comput., 8 (1979), pp. 463-478.

[HK73] J. E. HOPCROFT AND R. M. KARP, *An $n^{5/2}$ algorithm for maximum matchings in bipartite graphs*, SIAM J. Comput., 2 (1973), pp. 225-231.

[JvE86] C. T. M. JACOBS AND P. VAN EMDE BOAS, *Two results on tables*, Inform. Process. Lett., 22 (1986), pp. 43-48.

[Ma83] H. G. MAIRSON, *The program complexity of searching a table*, in Proc. 24th Annual Symposium on Foundations of Computer Science, Tucson, AZ, November 1983, pp. 40-47.

[Ma84] ———, *The program complexity of searching a table*, Ph.D. thesis, Department of Computer Science, Stanford University, Stanford, CA, 1984, STAN-CS83-988.

[Me82] K. MEHLHORN, *On the program size of perfect and universal hash functions*, in Proc. 23rd Annual Symposium on Foundations of Computer Science, Chicago, IL, 1982, pp. 170-175.

[Me84] ———, *Data Structures and Algorithms* 1: *Sorting and Searching*, Springer-Verlag, Berlin, Heidelberg, 1984.

[S89] A. SIEGEL, *On universal classes of fast hashfunctions, their time-space tradeoff, and their applications*, in Proc. 30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, NC, October 1989, pp. 20-25.

[SS80] J. P. SCHMIDT AND E. SHAMIR, *An improved program for constructing open hash tables*, in Proc. Automata Languages and Programming, Noordwijkerhout, Holland, July 1980 (ICALP80), pp. 569-581.

[SS89] J. P. SCHMIDT AND A. SIEGEL, *On aspects of universality and performance for closed hashing*, in Proc. 21st Annual Symposium on Theory of Computing, Seattle, WA, May 1989, pp. 355-366.

[SS90] ———, *The analysis of closed hashing under limited randomness*, in Proc. 22nd Annual Symposium on Theory of Computing, Baltimore, MD, May 1990, to appear.

[SvE84] C. SLOT AND P. VAN EMDE BOAS, *On tape versus core: An application of space efficient hash functions to the invariance of space*, in Proc. 16th Annual Symposium on Theory of Computing, Washington, DC, May 1984, pp. 391-400.

[TY79] R. E. TARJAN AND A. C. YAO, *Storing a sparse table*, Commun. Assoc. Comput. Mach., 22 (1978), pp. 606-611.

[Y81] A. C. YAO, *Should tables be sorted?*, J. Assoc. Comput. Mach., 28 (1981), pp. 615-628.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 787-804, October 1990

002

DOMINO GAMES AND COMPLEXITY*

ERICH GRÄDEL†

Der Mensch spielt nur,
wo er in voller Bedeutung des Wortes Mensch ist,
und er ist nur da ganz Mensch, wo er spielt.

FRIEDRICH SCHILLER
Über die ästhetische Erziehung des Menschen

Abstract. Domino games which describe computations of alternating Turing machines in the same way as dominoes (tiling systems) encode computations of deterministic and nondeterministic Turing machines are considered. The domino games are two-person games in the course of which the players build up domino tilings of a square of prescribed size. Acceptance of an alternating Turing machine corresponds to a winning strategy for one player—the number of moves in the game is the number of alternations of the Turing machine.

Let ATIME($T(n)$, m) denote the class of all sets that are accepted by some alternating Turing machine in time $T(n)$ with at most m alternations. It is shown that any problem in such a complexity class can be reduced to the strategy problem for some domino game. In particular domino games which are complete in the classes Σ_m^p and Π_m^p of the polynomial time hierarchy are found. This corresponds to the approach of Savelsbergh and van Emde Boas and of Lewis and Papadimitriou, who have shown that the theory of NP-completeness may also be founded on a finite domino problem instead of the satisfiability problem for propositional formulae.

Similar generalizations are possible for domino connectability problems: Domino thread games where the players build up a thread of dominoes are introduced; the first player wins if he can ultimately connect two given points. It is shown that a certain class of such games captures deterministic time complexity. In particular, games are found whose strategy problems are P-complete.

In the last section applications to the complexity of simple prefix classes in logical theories are briefly discussed.

Key words. domino problems, games, alternating Turing machines

AMS(MOS) subject classifications. 03D15, 68Q05, 68Q10, 68Q15

1. Introduction.

Domino problems have now been used for almost 30 years as a tool for proving undecidability or lower complexity bounds of various systems of propositional logics (see [12] for a survey) and of subclasses of the predicate calculus [11], [14], [16]. Usually a *domino system* is described as a finite set of *tiles* or *dominoes*, every tile being a unit square with fixed orientation and coloured edges; we have an unlimited supply of copies of every tile. A domino problem asks whether it is possible to tile a prescribed subset of the cartesian plane with elements of a given domino system, such that adjacent tiles have matching colours on their common edge and with perhaps some constraints on the tiles that are allowed on certain specific places (e.g., the origin).

Domino problems have turned out to be quite powerful for proving undecidability or lower complexity bounds essentially for two reasons. First, they are strong enough to encode the computations of Turing machines in a straightforward way: successive

* Received by the editors November 9, 1988; accepted for publication (in revised form) December 14, 1989. Sections 1-4 of this article were presented in preliminary form at the Symposium on Theoretical Aspects of Computer Science (STACS 88), held in February 1988 in Bordeaux, France. Part of this work was done while the author was staying at the University of Pisa, Italy, and was supported by the Swiss National Science Foundation.

† Mathematisches Institut, Universität Basel, Rheinsprung 21, CH-4051 Basel, Switzerland, email: graedel@urz.unibas.ch.

rows of the tiling represent successive configurations of the Turing machine. Thus we can design—by reduction of halting problems of one sort or another—domino problems that are undecidable or complete in some complexity class.

Second, dominoes have a very simple geometrical and combinatorial structure. Therefore reductions from domino systems to instances of other problems are easy to construct and simple in outcome. This makes them useful for proving the undecidability of "simple" classes of quantificational formulae. The most famous of these results is the undecidability of the $\forall\exists\forall$-class of the predicate calculus (without functions), established by Kahr, Moore, and Wang [14]. Refinements of this and extensions to other classes may be found in [11] and [16]. The economy of quantifiers that is possible in reductions from domino problems to first-order formulas can also be exploited in complexity theory. Fürer [8] has proved with this method that the (solvable) prefix class $\forall\exists \wedge \exists\exists$ of the predicate calculus has exponential complexity, and in [9] Grädel has shown that domino problems also yield good lower complexity bounds for subclasses of specific logical theories.

Since tilings of dominoes correspond to Turing machine computations and since domino problems ask about the *existence* of tilings they are particularly useful for nondeterministic complexity ("$\exists$ tiling" corresponds to "$\exists$ computation"). For *alternating* complexity classes *games* seem to be a more appropriate device than dominoes as has been pointed out by Harel [12]: "Domino problems are existential in nature and do not seem to extend in any natural way to capture alternation. . . . Thus while games are good for alternation, dominoes are good for single existentials."

But what sort of game should be chosen for describing alternating computations? Since every nonmathematician considers dominoes a game, why not also bring this into mathematics? In § 3 we define *domino games*[1] which generalize domino problems in the same way as alternating Turing machines generalize nondeterministic Turing machines: Given a domino system as above, two players, the *constructor* and the *saboteur*, build up a tiling of a square of size $T(n)$ (T a time resource function, n the length of some reasonable encoding of the domino system). The constructor wins if the square is at the end entirely—and correctly—tiled, otherwise the saboteur wins. The strategy problem of whether the constructor has a forced win in m moves is shown to be hard for $\bigcup_{c>0}$ ATIME($T(cn)$, m). It thus may serve as a starting point for reductions to other problems in order to show lower bounds of the form ATIME($T(cn)$, m).

The idea of introducing domino games and of translating alternating computations into their strategy problem is due to Chlebus [4]. His approach differs from ours in that Chlebus is not interested in bounding the number of moves (i.e., of the alternations of the Turing machines) to a fixed number, but in defining complete games for classes such as PSPACE, EXPTIME, and 2EXPTIME. The rules in his games are thus quite different from ours (the players alternate after every tile) and he obtains other sorts of applications.

Section 4 deals with the description of the polynomial-time hierarchy by domino games. Lewis and Papadimitriou [17] and Savelsbergh and van Emde Boas [19] have shown that the theory of NP-completeness may be founded on a bounded domino problem and that this approach has some advantages over the more common use of

[1] In the literature, the terms *domino systems*, *domino problems*, and *domino games* are sometimes used interchangeably. In this article we distinguish between them: A domino system is a set of dominoes, a domino problem asks after the existence of a tiling, and a domino game is a two-person game played with a domino system.

the satisfiability problem for propositional formulae. In § 4 we will generalize this to the polynomial-time hierarchy: we will find complete domino games for all levels Σ_m^p and Π_m^p of the hierarchy and also for PSPACE.

In §§ 5–7 we discuss other kinds of domino problems, e.g.: Given a domino system $\mathcal{D}$ and two points of the plane, it is asked whether there is a thread of dominoes from $\mathcal{D}$ connecting the two points. Such *domino connectability problems* have been considered by Myers [18] and Ebbinghaus [6], [7]; their results are reviewed in § 5. In § 6 we discuss generalizations to *domino thread games.* There are several possibilities. The first is a straightforward analogue of the domino (tiling) games treated in § 3. By introducing an additional restriction on the thread to be constructed we will define a second sort of domino thread game which will capture deterministic time complexity classes. In § 7 we treat several very simple variants of domino games whose strategy problems are complete for deterministic polynomial time. We conclude with a short discussion of applications.

2. Alternating Turing machines. The concepts of *alternating Turing machines* and of ATIME-complexity classes were introduced by Chandra, Kozen, and Stockmeyer. We recall here the definitions and some basic properties, following a slightly different approach which emphasizes from the beginning the close correspondence between alternating Turing machines and games. Further results are found, e.g., in [3].

An alternating Turing machine is a nondeterministic Turing machine whose set of states is partitioned into *existential, universal, accepting,* and *rejecting* states. For every input x the operation of M on x defines a *computation tree* $\mathcal{T}_{M,x}$ whose nodes are the configurations (the instantaneous descriptions); the root is the initial configuration of M on x and the children of every node are those configurations reachable via the transition relation of M in one step. Thus a computation of M on x corresponds to a path through $\mathcal{T}_{M,x}$ from the root to a leaf. A node of $\mathcal{T}_{M,x}$ is called existential, universal, accepting, or rejecting according to the attribute of the state belonging to it.

While we say in the nondeterministic case that M accepts x if and only if there is a path in the tree that leads to an accepting node, acceptance for alternating Turing machines is defined in a more complicated way: It is described best by a two-person game in the course of which players $\exists$ and $\forall$ define a path through the computation tree $\mathcal{T}_{M,x}$: at every existential node $\exists$ selects one of the children of the node, at universal nodes $\forall$ makes a similar move. If the path defined like this eventually leads to acceptance $\exists$ wins the game; otherwise $\forall$ wins. We say that M accepts x if and only if $\exists$ has a winning strategy for the game on $\mathcal{T}_{M,x}$. Moreover, we say that M accepts x in time t and with m alternations if such a winning strategy for $\exists$ still exists when all paths $\mathcal{T}_{M,x}$ longer than t nodes or with more than m changes of players automatically lead to rejection. It follows that nondeterministic Turing machines are just alternating Turing machines without universal states.

Let $T(n)$ and $A(n)$ be nondecreasing functions from natural to real numbers; we always assume that $1 \leqq A(n) \leqq T(n)$ for all n. The complexity class ATIME $(T(n), A(n))$ consists of the sets decided by an alternating Turing machine in time $T(n)$ and with $A(n)$ alternations. In analogy to deterministic and nondeterministic complexity classes we define:

$$\text{APTIME} = \bigcup_{k \in \mathbf{N}} \text{ATIME}(n^k, n^k), \qquad \text{AEXPTIME} = \bigcup_{k \in \mathbf{N}} \text{ATIME}(2^{n^k}, 2^{n^k}).$$

Where no bound on the number of alternations is imposed, i.e., if $A(n) = T(n)$ we usually omit $A(n)$ and write ATIME $(T(n))$ instead of ATIME $(T(n), T(n))$. It is

known [3], that for all functions $T(n) \geqq n$ and $S(n) \geqq \log n$:

$$\mathrm{ATIME}(T(n)) \subseteq \mathrm{DSPACE}(T(n)) \subseteq \bigcup_{c>0} \mathrm{ATIME}(c \cdot T(n)^2),$$

$$\mathrm{ASPACE}(S(n)) = \bigcup_{c>0} \mathrm{DTIME}(c^{S(n)}).$$

In particular, APTIME = PSPACE and ALOGSPACE = P.

3. Dominoes and domino games. Let S be $\mathbf{N} \times \mathbf{N}$ or a finite square $\{0, \cdots, m\} \times \{0, \cdots, m\}$. The classical domino problem, as described in the introduction may be formulated as follows:

Instance: $\mathcal{D}$ consisting of a finite set D and binary relations H, $V \subset D \times D$.

Question: Is there a tiling $\tau : S \to D$ such that for all $(x, y) \in S$

$$\tau(x, y) = d_i \ \wedge \ \tau(x+1, y) = d_j \ \Rightarrow \ (d_i, d_j) \in H,$$

$$\tau(x, y) = d_i \ \wedge \ \tau(x, y+1) = d_j \ \Rightarrow \ (d_i, d_j) \in V.$$

This formulation is equivalent to the more intuitive description by unit squares with coloured edges. In fact, H (respectively, V) just contain those pairs (d, d') of dominoes for which the right (upper) colour of d is equal to the left (lower) colour of d'. Conversely, given $\mathcal{D} = (D, H, V)$ as above, take a unit square tile for each triple (d, d', d'') with $(d, d') \in H$ and $(d, d'') \in V$ and colour its left and lower edge with d, its right edge with d', and its upper edge with d''.

A variant of this problem which is particularly convenient for the encoding of Turing machine computations is the *origin constrained domino problem.* We are given $\mathcal{D} = (D, D_0, H, V)$ where D, H, and V are as above, D_0 is a subset of D, and it is asked whether there is a tiling which places a tile from D_0 on the point $(0, 0)$. For $S = \mathbf{N} \times \mathbf{N}$ both problems are undecidable [23], [1].

We now generalize this to the notion of *domino games* which describe computations of alternating Turing machines in the same way as domino-tilings encode the computations of deterministic and nondeterministic Turing machines. Thus domino games will provide a convenient tool for proving lower bounds for ATIME-complexity classes.

We assume in the sequel that Turing machines and domino systems are encoded in a suitable way as strings over a finite alphabet and we identify them with their encodings. S_t denotes the square $\{0, \cdots, t\} \times \{0, \cdots, t\}$.

DEFINITION. A *domino game* $\langle \mathcal{D}, t \rangle$ is given by a domino system $\mathcal{D} = (D, D_0, H, V)$ where D is the disjoint union of two subsets E and A and D_0 is a subset of either E or A; tiles from E and A are called *existential* and *universal* tiles, respectively; t is a natural number specifying the size of the playboard.

The game is played by two persons, $\exists$ and $\forall$ also referred to as the *constructor* and the *saboteur.* The constructor tries to build a tiling of the square S_t, the saboteur wants to prevent him from achieving this goal. In the course of the game the players tile S_t, row after row, according to the following rules:

(1) Odd rows are tiled from the left to the right and even rows from the right to the left; so the game proceeds like a meander through S_t.

(2) The adjacency conditions imposed by H and V must be satisfied. If no player can place a next tile, the saboteur immediately wins.

(3) The constructor ($\exists$) uses the tiles from E, the saboteur ($\forall$) the tiles from A. A player moves—and has to move—until he cannot place a next tile. Then the other

player begins to move. Thus, whether D_0 is contained in E or in A determines which player has the first move.

(4) If S_t is entirely tiled, the constructor wins.

DEFINITION. Let GAME (t, m) denote the set of all dominoes $\mathcal{D}$ such that the constructor has a strategy to win the game $\langle \mathcal{D}, t \rangle$ in at most m moves; t may be a function of $|\mathcal{D}|$.

THEOREM 3.1. *Let M be an alternating Turing machine, and let $T(n)$ be a time constructible function. There is a polynomial reduction, taking every input $x = x_0 \cdots x_{n-1}$ to a domino $\mathcal{D}_x$ with length $O(n)$ such that for all m and for $t = T(n)$:*

$$M \text{ accepts } x \text{ in time } t \text{ with } m \text{ alternations} \Leftrightarrow D_x \in \text{GAME}\,(t+2, m).$$

Proof. There are well-known techniques to encode computations of deterministic and nondeterministic Turing machines into domino tilings in a very natural way: Successive rows of a tiling represent successive configurations of a computation. Here these techniques are modified such that the alternations of the Turing machine correspond to the changes of players in the domino game. In particular it must be assured that the moves of the saboteur do indeed correspond to valid steps of the Turing machine; he must be restrained from making moves which destroy this correspondence. Otherwise the saboteur could "cheat" by leading the game to a dead end for both players, which would not represent a rejecting configuration but a deviation from the intended course of the game.

(1) First we consider M as a nondeterministic machine and, for simplicity of notation, we assume that M has just one tape. Generalization to k-tape machines requires only minor modifications. To avoid "cheating" in the sense described above we first construct a domino system such that the following holds. If the first row is tiled in such a way that it encodes the initial configuration of M on input x, then every player who proceeds to place tiles in the appropriate order produces a tiling which corresponds to a computation of M on input x. It will be essential for our construction that odd rows are tiled from left to right and even rows from right to left.

If Q is the set of states and Σ the alphabet of the Turing machine, then every configuration can be described as a word over the alphabet $\Gamma := (Q \cup \{L, R, *\}) \times \Sigma$. The letter L or R that we attach to a symbol on the tape indicates whether it is at the left or the right side of the scanning head; this is to assure that there is exactly one element from $(Q \times \Sigma)$ in every configuration. The $*$ is for control purposes which will be explained below.

Informally the encoding idea is the following: the dominoes are triples $(a, b, c) \in \Gamma^3$ which encode a symbol in a configuration and its two neighbours; a configuration $\cdots a_{i-3}, a_{i-2}, a_{i-1}, (qa_i), a_{i+1}, a_{i+2}, \cdots$ is represented by

$$\cdots (La_{i-3}, La_{i-2}, La_{i-1})(La_{i-2}, La_{i-1}, qa_i)(La_{i-1}, qa_i, Ra_{i+1})(qa_i, Ra_{i+1}, Ra_{i+2}) \cdots .$$

Not all triples from Γ^3 are useful; e.g., in every configuration, to the right of a symbol $qa \in Q \times \Sigma$, only symbols from $\{R\} \times \Sigma$ may occur. So define, e.g., the set

$$LQR := \{(abc) \in \Gamma^3 \mid a \in \{L\} \times \Sigma,\ b \in Q \times \Sigma,\ c \in \{R\} \times \Sigma\}.$$

For every other triple A, B, C of letters from $\{Q, R, L, *\}$ a subset $ABC \subset \Gamma^3$ is defined in an analogous way. Furthermore we take to every such set ABC a disjoint copy $\overline{ABC}$.

The set of dominoes D is now the union of the following fourteen sets: LLL, $LL*$, LLQ, LQR, QRR, $*RR$, and RRR and their overlined copies.

Odd configurations are represented by "overlined" dominoes and even configurations by "normal" ones. Thus the adjacency conditions H and V force the horizontal neighbours to be of the same type, whereas vertical neighbours change their type from "normal" to "overlined" or vice versa. In addition H imposes that two adjacent dominoes have an overlap of two symbols as follows. First assume that both dominoes (abc) and $(a'b'c')$ are *not* contained in $LL*$ or $*RR$ (or their overlined copies); in this case the pair $(abc)(a'b'c')$ is in H if and only if $b = a'$ and $c = b'$. A domino from $LL*$, say $(L\alpha, L\beta, *\gamma)$, should be consic ered a "substitute" for either $(L\alpha, L\beta, L\gamma)$ or for some domino $(L\alpha, L\beta, q\gamma)$ (where ι is a state symbol). It will be played when a row is tiled from left to right at a position where the scanning head may move in from the right. In this case the overlap condition requires that a domino (abc) can be the right neighbour of $(L\alpha, L\beta, *\gamma)$ if $a = L\beta$ and if either $b = L\gamma$ or $b = q\gamma$ for some q. Analogous conditions hold for left neighbours of $*RR$-dominoes. In addition to the condition that "normal" and "overlined" dominoes alternate, the relation V forces

(i) the vertically adjacent dominoes to be compatible with the transition relation of the Turing machine (e.g., a domino $(\overline{L\alpha, q\beta, R\gamma})$ may have the upper neighbour $(q'\alpha, R\beta', R\gamma)$ if the transition relation contains the five-tuple $(q\beta q'\beta' L)$);

(ii) the $LL*$- and $*RR$-dominoes to be avoided whenever possible.

The following table indicates the possible upper neighbours (computations are described bottom up):

$$\left.\begin{array}{cccccc} & LQR & LQR & QRR & RRR & \\ LLQ & LLQ & LLQ & LQR & QRR & \\ LLL & LLL & LLL & LLQ & LQR & *RR \end{array}\right\} \text{possible upper neighbours,}$$

$$\begin{array}{cccccc} \overline{LLL} & \overline{LL*} & \overline{LQR} & \overline{LQR} & \overline{QRR} & \overline{RRR} \end{array}$$

$$\left.\begin{array}{cccccc} & \overline{LQR} & \overline{QRR} & \overline{RRR} & & \overline{LQR} \\ & \overline{LLQ} & \overline{LQR} & \overline{QRR} & \overline{QRR} & \overline{QRR} \\ \overline{LL*} & \overline{LLL} & \overline{LLQ} & \overline{LQR} & \overline{RRR} & \overline{RRR} \end{array}\right\} \text{possible upper neighbours.}$$

$$\begin{array}{cccccc} LLL & LLQ & LQR & QRR & RRR & *RR \end{array}$$

This produces the following. Assume that an odd row (e.g., the first) is tiled with dominoes of the types $\overline{LL*}$, $\overline{LQR}$, $\overline{QRR}$, $\overline{RRR}$, possibly $\overline{LLL}$ and $\overline{LLQ}$, but not $*RR$. Then the next row is even and tiled from right to left. Above the $\overline{RRR}$-dominoes, only $*RR$-dominoes can be placed (which means that the player does not know whether the head will move in from the left). But before the player reaches the points where he must place upper neighbours of $\overline{LL*}$-dominoes he must tile the points where the running head is encoded; hence he has already chosen the successor configuration and this leaves him no more liberty for the tiling of the rest of the row. In particular, the row will not contain any $LL*$-domino. In the next row the situation is symmetric: The player comes from the left and will encounter tiles "with incomplete information" (now $*RR$-tiles) only after he has crossed the square currently scanned by the Turing machine. Thus every row will contain dominoes with an asterisk only on one side of the running head and in the next row these dominoes will be met last so they cannot damage the game. Therefore the only liberty of the player lies in choosing one among several possible successor configurations.

The following diagram shows the possible tilings of an even row which follows an odd row with minimal possible information (i.e., containing no $\overline{LLL}$ and no $\overline{LLQ}$-dominoes):

LLL	LLQ	LQR	QRR	RRR	$*RR$	$*RR$	$\cdots$ row $2k$; head moves left
LLL	LLL	LLQ	LQR	QRR	$*RR$	$*RR$	$\cdots$ row $2k$; head does not move
LLL	LLL	LLL	LLQ	LQR	$*RR$	$*RR$	$\cdots$ row $2k$; head moves right
$\overline{LL*}$	$\overline{LL*}$	$\overline{LL*}$	$\overline{LQR}$	$\overline{QRR}$	$\overline{RRR}$	$\overline{RRR}$	$\cdots$ row $2k-1$.

For $\mathcal{D} = (D, H, V)$ defined in this way we conclude the following. *If we can arrange that the bottom row of a tiling encodes the initial configuration of M on input x, then the tiling produced during the game on S_t corresponds to a computation of a length t on input x.*

(2) There is a straightforward way to encode the initial configuration on x with $O(n)$ tiles. But this is not good enough: A representation of such a domino system by bit-matrices for H and V has length proportional to n^2. A more terse description can be given by listing explicitly the pairs in H and V; there are $O(n)$ such pairs so the total length of the list would be proportional to $n \log n$. But even that is not good enough: We want a representation of length $O(n)$.

For this purpose we define a more complicated encoding of the initial configuration by a domino system with $O(\sqrt{n})$ tiles which—using bit-matrices for H and V—can be represented by a string of length $O(n)$:

LEMMA. *The input configuration of M on an input x of length n can be encoded by the (unique) tiling of three rows with a domino system $\mathcal{X} = (X, X_0, H_x, V_x)$ with $O(\sqrt{n})$ dominoes. Moreover, $\mathcal{X}$ has the property that any player who begins by placing the unique element of X_0 at the origin and then proceeds as required by the rules of the game must produce in the third row a tiling which corresponds to the input configuration (via the encoding defined above).*

The input configuration on x is encoded by a word $x_0 x_1 \cdots x_n \flat^* \in D^*$ ($\flat$ is a domino which represents the blank symbol). Assume, for simplicity, that $n+1 = q^2$. We divide $x_0 \cdots x_n$ into segments of length q and encode each segment separately. The domino system $\mathcal{X} = (X, X_0, H_x, V_x)$ is constructed as follows:

X contains the dominoes:

— $(*)$, $(**)$, $(\flat)$ and $(\flat')$;

— (i) and $(\underline{i})$ for $i = 0, \cdots, q-1$;

— (a, i) and $(a, \underline{i})$ for $a \in D$, $i = 0, \cdots, q-1$.

Thus X consists of $O(q) = O(\sqrt{n})$ tiles. X_0 only contains the tile $(*)$.

H_x contains the pairs:

— $\langle *, \underline{1} \rangle$, $\langle **, 1 \rangle$;

— $\langle i, j \rangle$, $\langle \underline{i}, \underline{j} \rangle$ for $j = i+1$ in $\mathbf{Z}/q\mathbf{Z}$;

— $\langle (a, i), (a', i)\rangle$, $\langle (a, \underline{i}), (a', i)\rangle$ and $\langle (a, i-1), (a', \underline{i})\rangle$ for every i and every $a, a' \in D$;

— $\langle \flat, \flat'\rangle$, $\langle \flat', \flat'\rangle$ and $\langle (a, q-1), \flat\rangle$ for every $a \in \mathscr{D}$.

V_x contains the pairs:

— $\langle *, **\rangle$, $\langle **, (x_0, \underline{0})\rangle$, $\langle 0, \flat\rangle$;

— $\langle \underline{i}, i\rangle$ and $\langle i, \flat'\rangle$ for $i = 0, \cdots, q-1$;

— $\langle i, (a, j)\rangle$ for which $i \neq 0$ and $x_{jq+i} = a$;

— $\langle 0, (a, \underline{j})\rangle$ for which $x_{jq} = a$.

It is easily verified that when we begin by placing the domino $(*)$ at the origin and then proceed in tiling the first row from left to right, the second from right to left, and the third again from left to right, then we have at every point one and only one possibility to place a domino and we will produce a rectangle of the following form:

x_0	x_1	$\cdots$	x_{q-1}	x_q	x_{q+1}	$\cdots$	x_{jq+i}	$\cdots$	x_n	$\flat$	$\flat'$	$\cdots$
$\underline{0}$	0		0	$\underline{1}$	1		j		$q-1$			
$**$	1	$\cdots$	$q-1$	0	1	$\cdots$	i	$\cdots$	$q-1$	0	1	$\cdots$
$*$	$\underline{1}$	$\cdots$	$\underline{q-1}$	$\underline{0}$	$\underline{1}$	$\cdots$	$\underline{i}$	$\cdots$	$\underline{q-1}$	$\underline{0}$	$\underline{1}$	$\cdots$

This proves the lemma. So we are able to encode the computations of length t of M (considered as a nondeterministic Turing machine) on input x by the tilings of the square S_{t+2} by a domino system of length $O(n)$ that we obtain by putting together $\mathscr{D}$ and $\mathscr{X}$ in an appropriate way.

(3) Now we want to capture the alternations of M by changes of players in a domino game. The states of M are partitioned into existential, universal, accepting, and rejecting states. With each state q of M we can associate some of the dominoes in $\mathscr{D}$, namely, those triples (a, b, c) for which a, b, or c is equal to $q\alpha$ (for some $\alpha \in \Sigma$).

We now modify $\mathscr{D} = (D, H, V)$ in the following way. Take two distinct copies E and A of D, one for the existential, one for the universal player, and set $\tilde{\mathscr{D}} = E \cup A$. $\tilde{V}$ contains all pairs (d, d') that were contained in V except if d and d' are both in $E(A)$ and d is associated to a universal (existential) state; $\tilde{H}$ contains all pairs that were in H. In other words, if the existential player places a domino associated to a universal state, then no other tile from E can be put above it, and so his move will be terminated in the next row. Similarly, the move of the universal player ends with laying a domino associated to an existential state.

Finally, we merge in a suitable way $\tilde{\mathscr{D}}$ and $\mathscr{X}$ into one domino system $\mathscr{D}_x$. $\mathscr{X}$ is included in E if the initial state of M is existential, otherwise it is included in A.

(4) Because $T(n)$ is a time constructible function we may assume that the accepting computations remain after acceptance in the accepting configuration, whereas the rejecting computations of M on inputs of length n reject after at most $T(n)-1$ steps. Thus the computations of length t are precisely the accepting computations.

When the domino game $\langle \mathscr{D}_x, t\rangle$ is played, the two players build up a tiling of S_{t+2}, which corresponds to one particular computation of M on x. In the game a move of either player is terminated in one of the three following situations:

(a) The constructor has placed a tile that is associated to a universal state or the saboteur has placed a tile associated to an existential state; this corresponds

to a change from an existential to a universal configuration or vice versa of the Turing machine.

(b) The space is entirely tiled, i.e., the constructor has won. This corresponds to a computation of length t which is by the remark above an accepting computation.

(c) No player can move anymore, but S_{t+2} is not yet fully tiled. Thus, the saboteur has won. This corresponds to a computation shorter than t, hence a rejecting computation of M on x.

So the constructor wins (a run of) the game if the corresponding computation is accepting and the number of moves in the game is equal to the number of alternations of M. Hence the constructor has a winning strategy in m moves for the game $\langle \mathscr{D}_x, t+2\rangle$ if and only if M accepts x (in the sense of the definition given in § 2) in time t with m alternations. □

COROLLARY 3.2. *Let $T(n)$ be a time constructible function such that for some $d>0$, $T(dn)\log(T(dn)) = o(T(n))$. Then there exists a positive constant c such that*

$$\text{GAME}(T(n), m) \notin \text{ATIME}(T(cn), m).$$

Proof. If T fulfills the required condition, then there exists a set $L \in \text{ATIME}(T(n), m) - \text{ATIME}(T(dn), m)$. Let M be an alternating Turing machine deciding L in time $T(n)$ with m alternations. If $\mathscr{D}_x$ is the domino constructed from M and x ($|x| = n$) in Theorem 3.1, then

$$\mathscr{D}_x \in \text{GAME}(T(n), m) \quad \text{iff } x \in L.$$

The construction of $\mathscr{D}_x$ requires time bounded by n^k and has length less than or equal to kn for a constant k. Choose a $c>0$ with $T(ckn)+n^k < T(dn)$ for large n. If GAME $(T(n), m)$ were in ATIME $(T(cn), m)$ we had $L \in \text{ATIME}(T(dn), m)$, contradicting the assumption. □

Note that, in general, ATIME-classes with alternations bounded by a constant probably do not contain complete sets. Indeed we have the decomposition

$$\text{ATIME}(T(n), m) = \Sigma_m\text{-TIME}(T(n)) \cup \Pi_m\text{-TIME}(T(n))$$

where Σ_m-TIME $(T(n))$ is the restriction of ATIME $(T(n), m)$ to sets accepted by alternating machines whose initial state is existential; Π_m-TIME $(T(n))$ is defined analogously for universal initial states. If a set L were complete in, say, $\bigcup_{c>0}$ ATIME $(T(cn), m)$, then either $L \in \bigcup_{c>0} \Sigma_m$-TIME $(T(cn))$ or $L \in \bigcup_{c>0} \Pi_m$-TIME $(T(cn))$. Because L is complete in $\bigcup_{c>0}$ ATIME $(T(cn), m)$, both cases would imply that $\bigcup_{c>0} \Sigma_m$-TIME $(T(cn))$ is closed under complement which is generally believed to be false for most functions $T(n)$.

So, in particular, GAME $(T(n), m)$ is probably not $\bigcup_{c>0}$ ATIME $(T(cn), m)$-complete. However we can decompose GAME $(T(n), m)$ into two subsets according to which player has the first move:

$$\text{CONSTRUCTOR-GAME}(T(n), m) = \{\mathscr{D} \in \text{GAME}(T(n), m) \mid D_0 \subseteq E\},$$

$$\text{SABOTEUR-GAME}(T(n), m) = \{\mathscr{D} \in \text{GAME}(T(n), m) \mid D_0 \subseteq A\}.$$

Theorem 3.1 proves that these two sets are hard for Σ_m-TIME $(T(n))$ and Π_m-TIME $(T(n))$, respectively. The following result gives almost matching upper bounds.

THEOREM 3.3. *If $T(n)$ is a time constructible function, then*

$$\text{CONSTRUCTOR-GAME}(T(n), m) \in \Sigma_m\text{-TIME}(nT(n)^2),$$

$$\text{SABOTEUR-GAME}(T(n), m) \in \Pi_m\text{-TIME}(nT(n)^2).$$

Proof. We can design an alternating Turing machine which, given a domino game $\mathcal{D}$, constructs the tree of all possible runs of the game with at most m moves and verifies whether the constructor has a winning strategy. Every branch of the tree represents a possible sequence of moves, i.e., simulates the covering of (at most) $T(n)^2$ points. At each of these points, one tile from D must be chosen and it must be verified that the rules of the game are satisfied; in particular it must be checked that the adjacency conditions are fulfilled: if the alternating machines has two working heads this can be done in time $O(n)$. Therefore the entire procedure takes $O(nT(n)^2)$ steps and m alternations. The initial state of the machine is existential for CONSTRUCTOR-GAME $(T(n), m)$ and universal for SABOTEUR-GAME $(T(n), m)$. By the Linear Speedup Theorem the theorem follows. □

Observe, that for functions $T(n)$ with at least exponential growth there is a constant c such that $nT(n)^2$ is dominated by $T(cn)$. Thus for functions $T(n)$ with at least exponential growth

CONSTRUCTOR-GAME $(T(n), m)$ is $\bigcup_{c>0} \Sigma_m$-TIME $(T(cn))$-complete, and

SABOTEUR-GAME $(T(n), m)$ is $\bigcup_{c>0} \Pi_m$-TIME $(T(cn))$-complete.

4. Domino games and the polynomial-time hierarchy. In the textbook of Lewis and Papadimitriou [17] as well as in an article by Savelsbergh and van Emde Boas [19] a bounded tiling problem is introduced which is shown to be a viable alternative to SATISFIABILITY as a foundation of the theory of NP-completeness. In particular, if we are interested in the NP-completeness of combinatorial problems arising in computational practice such as EXACT COVER, KNAPSACK, TRAVELING SALESPERSON, or LINEAR INTEGER PROGRAMMING, it seems more elegant to define direct reductions from bounded domino problems, avoiding the detour over the (in this context) rather unnatural satisfiability problem.

The only difference to the approach in § 3 is that we now include the size of the square to be tiled as part of the instance of the problem. The size of the square is given in unary notation.

DEFINITION. SQUARE TILING $:= \{(\mathcal{D}, 1^n) \mid \mathcal{D}$ tiles $S_n\}$.

THEOREM 4.1 [17], [19]. SQUARE TILING *is* NP*-complete.*

Domino games allow a generalization to the whole polynomial time hierarchy as follows (see [21] for definitions and basic results on the polynomial time hierarchy).

DEFINITION. DOMINO GAME $(m, \exists)$ and DOMINO GAME $(m, \forall)$ consist, respectively, of the pairs $(\mathcal{D}, 1^n)$ such that $\exists$, respectively, $\forall$ has the first move in the game defined by $\mathcal{D}$ on an $n \times n$ playboard, and such that the constructor has a winning strategy in m moves; thus, with the notation of § 3:

$$\text{DOMINO GAME}(m, \exists) = \{(\mathcal{D}, 1^n) \mid \mathcal{D} \in \text{GAME}(n, m) \text{ and } D_0 \subseteq E\},$$

$$\text{DOMINO GAME}(m, \forall) = \{(\mathcal{D}, 1^n) \mid \mathcal{D} \in \text{GAME}(n, m) \text{ and } D_0 \subseteq A\}.$$

Note that DOMINO GAME $(1, \exists)$ = SQUARE TILING.

THEOREM 4.2. *For all* $m \geqq 1$:

DOMINO GAME $(m, \exists)$ *is* Σ_m^p*-complete,*

DOMINO GAME $(m, \forall)$ *is* Π_m^p*-complete.*

Proof. It is clear that DOMINO GAME $(m, \exists)$ and DOMINO GAME $(m, \forall)$ are contained in Σ_m^p and Π_m^p, respectively. Completeness follows from Theorem 3.1. If L is in Σ_m^p (Π_m^p), then it is decided by an alternating Turing machine M with m alternations in time $p(n)$ (p a polynomial), beginning in an existential (universal) state. Hence,

any input x may be mapped in polynomial time to a domino $\mathcal{D}_x$ and a string $1^{p(n)}$ such that $(\mathcal{D}_x, 1^{p(n)})$ is in DOMINO GAME $(m, \exists)$ (DOMINO GAME $(m, \forall)$) if and only if $x \in L$. □

Without restriction of the number of moves we obtain a PSPACE-complete problem. Let SQUARE DOMINO GAME denote the set of pairs $(\mathcal{D}, 1^n)$ such that the constructor has a winning strategy (with arbitrary number of moves) for the tiling of a $n \times n$-square with $\mathcal{D}$. Because PSPACE = APTIME we infer that SQUARE DOMINO GAME is PSPACE-complete. A similar theorem has also been proved by Chlebus [4].

5. Domino connectability problems. Given a domino system $\mathcal{D}$ we can not only ask whether it is possible to tile a given space S by $\mathcal{D}$ but also whether two given points can be connected by a thread of dominoes. Such *domino connectability problems* have been investigated by Myers [18] and Ebbinghaus [6], [7]. Let us shortly review their definitions and results.

DEFINITION. Let S_{mn} be the rectangle $\{0, \cdots, m\} \times \{0, \cdots, n\}$. A *thread* Θ in S_{mn} is a finite sequence $s_1, \cdots, s_l$ of distinct elements of S_{mn} such that s_i and s_j have a common edge if and only if $j = i+1$ or $i = j+1$.

Let $\mathcal{D}$ be a domino system as defined in § 3. A *$\mathcal{D}$-thread* is a tiling $\tau : \Theta \to \mathcal{D}$ of a thread Θ by $\mathcal{D}$ (satisfying the adjacency conditions imposed by H and V). Θ is the *support* of τ.

There are various undecidable connectability problems.

THEOREM 5.1 (Ebbinghaus [6]). *Let (n, m) be a fixed point in $\mathbf{N} \times \mathbf{N}$, not adjacent to $(0, 0)$. Then the following two problems are undecidable*:

(1) *Given a domino system $\mathcal{D}$, decide whether there exists a $\mathcal{D}$-thread in $\mathbf{Z} \times \mathbf{N}$ from $(0, 0)$ to (n, m).*

(2) *Given a domino system $\mathcal{D}$ with a distinguished element d, decide whether there exists a $\mathcal{D}$-thread τ from $(0, 0)$ to (n, m) with $\tau(0, 0) = d$.*

In fact these problems are Σ_1^0-complete. Problems with higher degrees of unsolvability are obtained by considering infinite threads. For instance, the problem of deciding whether a domino system $\mathcal{D}$ with a distinguished element d allows a $\mathcal{D}$-thread τ from $(0, 0)$ to infinity with $\tau(0, 0) = d$ is Π_1^0-complete and if we impose the additional requirement that the support of τ is not finally a straight line, then we obtain even a Σ_1^1-complete problem (see [7]).

If we drop the restrictions that the $\mathcal{D}$-thread from $(0, 0)$ to (n, m) must start with a distinguished tile d or that it must lie entirely in the upper halfplane we no longer have undecidability:

THEOREM 5.2 (Myers) [18].

$\{(\mathcal{D}, 1^n, 1^m) \mid \exists$ *$\mathcal{D}$-thread from $(0, 0)$ to (n, m)*$\}$ *is* PSPACE-*complete.*

Now we shift our attention to connectability problems in finite rectangles S_{mn}. In analogy to SQUARE TILING and RECTANGLE TILING (see, e.g., [19]) which are NP, respectively, PSPACE-complete we have Theorem 5.3.

THEOREM 5.3 (Ebbinghaus [7]).

$\{(\mathcal{D}, 1^n) \mid \exists$ *$\mathcal{D}$-thread from $(0, 0)$ to $(0, n)$ in S_{nn}*$\}$ *is* NP-*complete,*

$\{(\mathcal{D}, 1^n) \mid \exists$ *$\mathcal{D}$-thread from $(0, 0)$ to $(0, n)$ in some S_{mn}*$\}$ *is* PSPACE-*complete.*

The general idea of the proof is due to Myers. A Turing machine computation is represented by a thread which winds through the rectangle. The instantaneous descriptions (ID's) are encoded by the *geometry* of subthreads which go from left to right at a distance of six and which are separated by *transition threads* going from right to left.

The domino system is constructed in such a way that the only valid vertical arrangements of a transition thread between two ID-threads (with respect to the requirement that each domino has a common edge only with its immediate predecessor and successor) are those allowed by the transition relation of the Turing machine. Furthermore, it is arranged that an ID-thread is connectable with the point $(0, n)$ if and only if it encodes an accepting configuration. For details see [18] and [7].

6. Domino thread games. Exactly as domino tilings were generalized to two-person domino games which describe alternating computations we could define connectability games in the following straightforward way: The game is given by a domino system $\mathcal{D}$ with tiles partitioned into two subsets E and A, and by a number t specifying the size of the playboard. The two players, the constructor and the saboteur, build up a domino thread from $(0, 0)$. Each player moves until he cannot place a next tile; then his opponent begins to move. The constructor wins the game when a connection from $(0, 0)$ to (t, t) in S_t is established; otherwise the saboteur wins.

Obviously the set of domino systems $\mathcal{D}$ for which the constructor has a winning strategy in m moves for the game defined by $\mathcal{D}$ in this way on the playboard of size $T(|\mathcal{D}|)$ is hard for ATIME $(T(n), m)$ (for $T(n)$ satisfying the conditions of the previous theorems).

So this gives nothing that is really new; it is just a combination of the techniques of Myers and Ebbinghaus with the idea of § 3.

Therefore we prefer to look at a different class of domino thread games, which will capture *deterministic*-time complexity classes. The goal of the constructor is to connect $(0, t)$ with $(0, 0)$ by a $\mathcal{D}$-thread which never goes upwards, only downwards, to the left or to the right. Thus, if either player puts a tile on the point (x, y) then in the next move his opponent must place a tile at one of the points $(x-1, y)$, $(x, y-1)$, or $(x+1, y)$.

DEFINITION. A *domino thread game* is a given by a pair $\langle \mathcal{D}, t\rangle$, where $\mathcal{D} = (D, D_0, H, V)$ is a domino system and t is a natural number. It is played on the square $\{0, \cdots, t\} \times \{0, \cdots, t\}$ by two players, the constructor and the saboteur, according to the following rules:

(1) The constructor begins by placing a domino $d_0 \in D_0$ on $(0, t)$.

(2) The players alternately select a tile from $\mathcal{D}$ and put it at the right, left or lower (not the upper!) edge of the last tile placed by the opponent, such that it has no common edge with any other tile and such that the adjacency conditions are satisfied.

(3) The constructor wins the game when a thread is constructed that connects $(0, t)$ to $(0, 0)$; otherwise the saboteur wins.

DEFINITION. Let THREAD GAME $(T(n))$ denote the set of domino systems $\mathcal{D}$ such that the constructor has a forced win for the thread game defined by $\mathcal{D}$ on the square of size $T(|\mathcal{D}|)$.

THEOREM 6.1. *Let $T(n)$ be a function from* $\mathbf{N}$ *to* $\mathbf{N}$ *such that $T(n) \geqq n$ for all n. Then*

$$\text{THREAD GAME}\,(T(n)) \in \bigcup_{k>0} \text{DTIME}\,(T^k(n)).$$

Proof. We show that THREAD GAME $(T(n))$ is contained in $\bigcup_{c>0}$ ASPACE $(\log T(n))$ (which is equal to $\bigcup_{k>0}$ DTIME $(T(n)^k)$ by Theorem 3.3 in [3]). In fact, to simulate a run of the game defined by $\mathcal{D}$, the alternating Turing machine must only "keep in mind" the coordinates of the last point that was tiled, which tile was placed, and the directions of the last two moves (down, right or left); this is because the next move may go to the right (left) if and only if none of the last two moves went to the left (right). Moves that go downwards are never forbidden.

Thus the simulation requires space $O(\log T(n))$. □

For proving a corresponding lower bound, we must encode the computations of a deterministic Turing machine M into the game. Let Q and Σ be the set of states and the alphabet of M and set $\Gamma := \Sigma \cup (Q \times \Sigma)$. With M we can associate a function $F: \Gamma^3 \to \Gamma$ such that the computation of M on input $x = x_0, \cdots, x_{n-1}$ is described by a $t \times t$-matrix (the computation table) with entries from Γ; t is the length of the computation and the elements $C(i, j)$ of the matrix satisfy:

$C(i, j+1) = F(C(i-1, j), C(i, j), C(i+1, j))$;

$C(0, 0) = (q_0, x_0)$, $C(i, 0) = x_i$ for $0 < i < n$ and $C(i, 0) = \flat$ (blank) for $i \geqq n$.

THEOREM 6.2. *Let M be a deterministic Turing machine. There is a polynomial reduction, taking every input $x = x_0 x_1 \cdots x_{n-1}$ to a domino system $\mathcal{D}_x$ of length $O(n \log n)$ such that for all $t \in \mathbf{N}$:*

$$M \text{ accepts } x \text{ in time } t \Leftrightarrow \mathcal{D}_x \in \text{THREAD GAME}(2t).$$

Proof. For any input x consider the following game G_x. Player $\exists$ wants to prove that M accepts x and states that M enters the accepting state at time t while scanning symbol a at position i. In terms of the computation table, he claims that $C(i, t) = (q_a, a)$. To prove this he indicates a triple $(\gamma_{-1}, \gamma_0, \gamma_1) \in \Gamma^3$ such that $F(\gamma_{-1}, \gamma_0, \gamma_1) = (q_a, a)$ and states that $C(i-1, t-1) = \gamma_{-1}$, $C(i, t-1) = \gamma_0$ and $C(i+1, t-1) = \gamma_1$. Player $\forall$ doubts this: he chooses m from $\{-1, 0, 1\}$ and challenges $\exists$ to justify that $C(i+m, t-1) = \gamma_m$. $\exists$ answers by selecting a new triple $(\gamma'_{-1}, \gamma'_0, \gamma'_1) \in F^{-1}(\gamma_m)$ for $C(i+m-1, t-2)$, $C(i+m, t-2)$ and $C(i+m+1, t-2)$. Then he is challenged again by $\forall$ to prove one of the three claims, and so on. The game is terminated when the last row of the computation table is reached. $\exists$ wins if and only if his last challenged claim for, say, $C(j, 0)$, is true (which is easily verified). Obviously, $\exists$ has a winning strategy for G_x (namely, to make always correct claims) if M accepts x. Conversely, if M does not accept x, $\forall$ can win the game by always challenging one of the false claims of $\exists$.

This game is implicit in the proof due to Chandra, Kozen, and Stockmeyer [3] that DTIME $(T(n)) \subseteq$ ASPACE $(c \log T(n))$ for all $T(n) \geqq n$ and all $c > 0$. We define a domino system $\mathcal{D}_x$ such that G_x corresponds (almost) step by step to the thread game defined by $\mathcal{D}_x$. The size of the board will be $2t$ and the thread that is constructed in the course of the game will consist of three subthreads:

(1) A thread from $(0, 2t)$ to $(2i, 2t')$ which ends with a tile (q_a, a), placed by the constructor $(\exists)$. This subthread represents $\exists$'s claim that M enters the accepting state at time $t' \leqq t$ at position i, reading tape symbol a. Three tiles are enough to enforce this beginning.

(2) The main part of the thread goes from $(2i, 2t')$ to a point $(2j, 0)$ in the bottom row and represents the actual game G_x. The set of dominoes includes Γ^3 and $\Gamma \times \{\exists, \forall, \forall'\}$. Suppose the constructor has placed a tile (α, β, γ) from Γ^3 at the point $(2i, 2j)$ below a tile $(\delta_\forall) \in \Gamma \times \{\forall\}$; by the construction of the domino system this is only possible if $F(\alpha, \beta, \gamma) = \delta$ and represents the claims in G_x that $C(i, j-1) = \alpha$, $C(i, j) = \beta$ and $C(i+1, j) = \gamma$. The saboteur $(\forall)$ now can place either $(\alpha_\forall)$ at the left side, $(\beta_\forall)$ at the lower side or $(\gamma_\forall)$ at the right side of (α, β, γ); each of these possibilities represents the corresponding challenge in G_x. Assume that the saboteur has placed $(\alpha_\forall)$. The constructor puts $(\alpha_\exists)$ at the left side of $(\alpha_\forall)$, which leaves the saboteur the only possibility to place $(\alpha_{\forall'})$ below it; if the saboteur has gone to the right (challenging γ) symmetric moves are done (but no such moves are necessary, if the saboteur has challenged β). Now we are in the same situation as at the beginning: The constructor must place a new triple $(\alpha', \beta', \gamma')$ at the point $(2(i+m), 2(j-1))$ $(m \in \{-1, 0, 1\})$ to

meet the challenge of the saboteur. This is repeated until the last row is reached. The number of tiles to represent this part of the game depends only on M (not on x). The thread reaches $(0, 0)$ if and only if β is, in fact, the jth symbol of the input configuration. The domino $(\beta, j, *)$ is a copy of (β, j) with the exception that it allows no adjacent dominoes on its right edge.

(3) The last part of the thread goes from $(2j, 0)$ to $(0, 0)$. Here the players have no more free choices: the thread can be terminated to $(0, 0)$ if and only if the tile placed on $(2j, 0)$ corresponds to the jth symbol in the initial configuration of M on x, i.e., if the last claim of the constructor was correct. It is straightforward to encode this part with $O(n)$ tiles such that H and V also have cardinality $O(n)$; therefore the domino system may be encoded by a string of length $O(n \log n)$.

Thus, the constructor has a winning strategy for $\langle \mathscr{D}_x, 2t \rangle$ if and only if M accepts x in time t. The explicit construction of $\mathscr{D}_x$ is left to the reader. □

COROLLARY 6.3. *Let T be a time-constructible function such that there is a $d > 0$ with $T(dn) \log (T(dn)) = o(T(n))$. Then there is a constant $c > 0$ with*

$$\text{THREAD GAME}\,(T(n)) \notin \text{DTIME}\,(T(cn/\log n)).$$

Proof. If T_1 and T_2 are two time-constructible functions such that $T_1(n) \log (T_1(n)) = o(T_2(n))$, then there is a language L decidable in time $T_2(n)$ but not in time $T_1(n)$. We thus can find an $L \in \text{DTIME}\,(T(n)) - \text{DTIME}\,(T(dn))$. By Theorem 6.2 every input x is reducible in polynomial time to a domino system $\mathscr{D}_x$ of length $O(n \log n)$, which is in THREAD GAME $(T(n))$ if and only if $x \in L$. □

7. *P*-complete domino games. From Theorems 6.1 and 6.2, together with the observation that the reduction in Theorem 6.2 can actually be done in logarithmic space, it follows that THREAD GAME (n) is P-complete. To obtain an analogous notation with SQUARE TILING and SQUARE DOMINO GAME we can define SQUARE THREAD GAME to be the set of pairs $(\mathscr{D}, 1^n)$ such that the constructor has a winning strategy for the thread game $\langle \mathscr{D}, n \rangle$. Certainly SQUARE THREAD GAME is P-complete.

There are, however, even simpler possibilities for defining P-complete domino games. These games are purely one-dimensional: we have instead of H and V only one adjacency relation A and the distinction between tiling games and thread games thus becomes meaningless.

DEFINITION. We consider three sorts of *one-dimensional domino games* G_1, G_2, and G_3. All are based on a one-dimensional domino system, i.e., a binary relation A on a finite set of tiles D. The players alternately place a tile next to the last tile placed by the opponent such that the adjacency relation A is satisfied. The constructor begins the game. In addition we have the following rules:

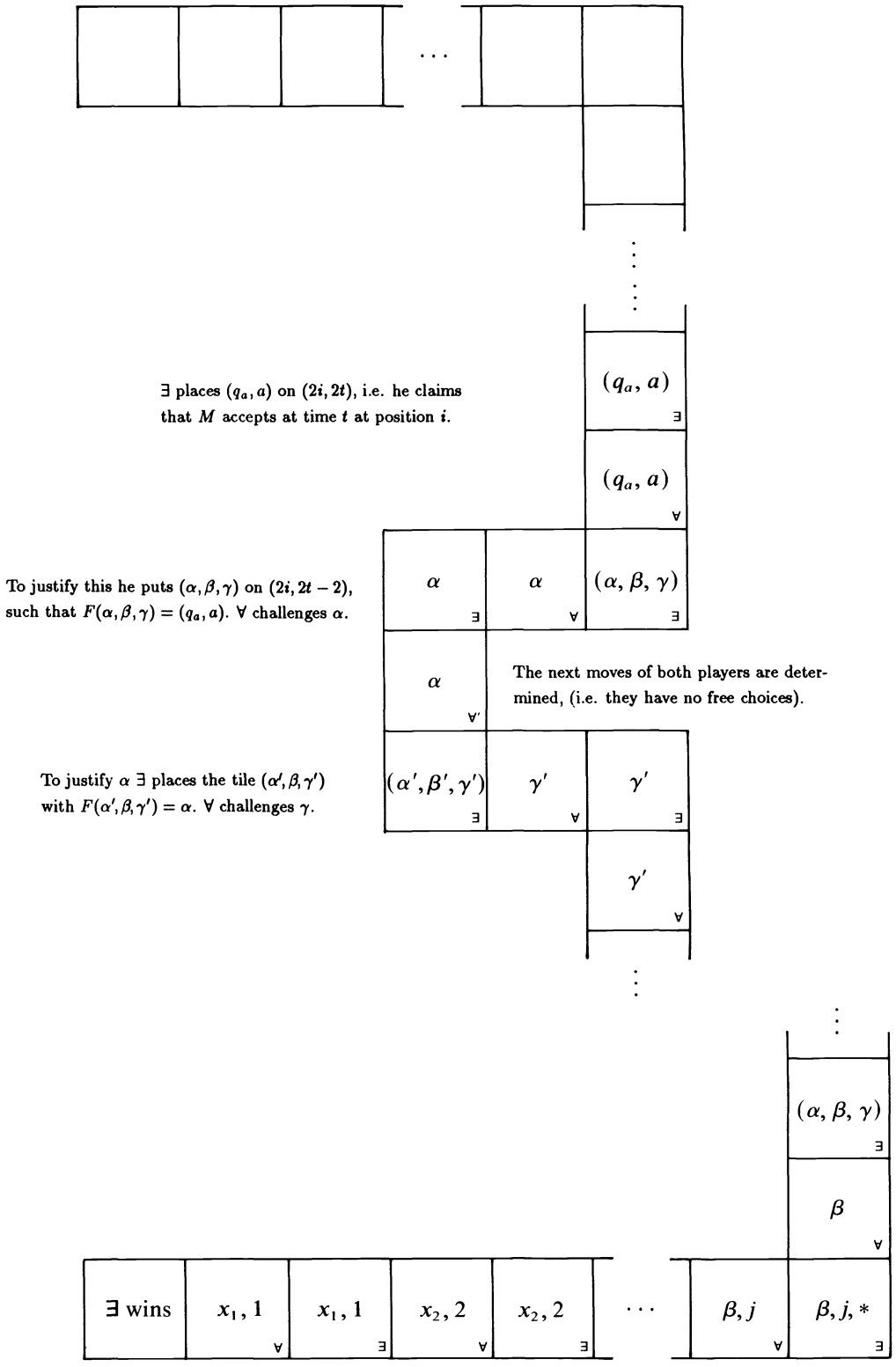
G_1: Given a one-dimensional domino system (D, A) and subsets D_0 and D_1 of D. The constructor wins if he accomplishes a tiling that begins with a tile from D_0 and ends with a tile from D_1.

G_2: Given D, A, and D_0 as above, but instead of D_1 a number 1^n in unary notation. The constructor wins if he can construct a tiling of length n beginning with an element of D_0.

G_3: The same as G_2 but without initial constraint: Given D, A, and 1^n; the constructor wants to build a tiling of length n.

For $i = 1, 2, 3$ denote by G_i those instances of the corresponding game for which the constructor has a winning strategy.

THEOREM 7.1. *G_1, G_2, and G_3 are P-complete.*


∃ places (q_a, a) on $(2i, 2t)$, i.e. he claims that M accepts at time t at position i.
To justify this he puts (α, β, γ) on $(2i, 2t-2)$, such that $F(\alpha, \beta, \gamma) = (q_a, a)$. ∀ challenges α.
The next moves of both players are determined, (i.e. they have no free choices).
To justify α ∃ places the tile $(\alpha', \beta, \gamma')$ with $F(\alpha', \beta, \gamma') = \alpha$. ∀ challenges γ.
(q_a, a) ∃
(q_a, a) ∀
α ∃
α ∀
(α, β, γ) ∃
α ∀'
$(\alpha', \beta', \gamma')$ ∃
γ' ∀
γ' ∃
γ' ∀
(α, β, γ) ∃
β ∀
∃ wins
$x_1, 1$ ∀
$x_1, 1$ ∃
$x_2, 2$ ∀
$x_2, 2$ ∃
β, j ∀
$\beta, j, *$ ∃

Proof. All three games are in P, because they can be simulated by an alternating Turing machine in logarithmic space. To prove completeness we reduce the problem GEN to each of the three games. GEN was defined and proved to be P-complete by Jones and Laaser [13]:

Instance: A set X, a binary operation $\cdot$ on X, a subset $S \subseteq X$, and an element $x \in X$.

Question: Is x contained in the set $\langle S \rangle$ generated by S under $\cdot$?

We first reduce GEN to G_1. Set:

$$D = X \dot{\cup} (X \times X), \quad D_0 = \{(a, b) \mid a \cdot b = x\}, \quad D_1 = S,$$

$$A = \{(a, (b, c)) \mid b \cdot c = a\} \cup \{((b, c), a) \mid a = b \vee a = c\}.$$

Obviously this defines a log-space reduction. Furthermore, if $x \in \langle S \rangle$ the constructor wins by answering any move a of the saboteur with a pair (b, c) such that $b \cdot c = a$ and such that b and c have shorter representations than a as a product of elements of S. Conversely, if $x \notin \langle S \rangle$ the saboteur wins by always choosing that element of a pair (b, c) not generated by S.

Observe that if x is generated by S, then it can be written as a term with depth over S at most $|X|$. So the winning strategy of the constructor requires at most $2|X|$ moves of both players together. Set $n := 2|X|+1$ and let D, D_0, and A be as above. First we add to A all pairs of elements of S, i.e., we set $A' = A \cup (S \times S)$. Now we can define the reduction G_2: $\tilde{D} := D \times \{1, \cdots, n\}$, $\tilde{D}_0 := D_0 \times \{1\}$, and $\tilde{A}$ contains those pairs $(d, i-1)$; (d', i) for which either $(d, d') \in A'$ and $i < n$ or for which $i = n$ and d, d' are both in S.

Thus $(\tilde{D}, \tilde{D}_0, \tilde{A}, 1^n)$ allows a winning strategy to build a tiled thread of length n if and only if (D, D_0, D_1, A) allows a winning strategy for a tiling ending with a tile in $D_1 = S$.

Finally, note that the initial constraint is redundant, since every winning strategy for the constructor requires that he begins with a tile indexed with one. Thus we also have a reduction to G_3. Hence all three games are P-complete. □

Remark. One-dimensional domino games were independently introduced by Toràn in his doctoral thesis [22]. He defines games $R(m)$ consisting of 2^{mn} tiles, each tile being a pair of colours coded by a string of length $2mn$; Toràn shows that the problem of whether the constructor has a strategy for tiling a row of length $m+1$ is complete in the class Σ^l_m of the *logarithmic time hierarchy.* (The logarithmic time hierarchy was defined in [3] by logarithmic time bounded alternating Turing machines with indirect access.) Toràn also found a P-complete domino game which is similar to our game G_1.

8. Applications.

> Nessuno ha mai sostenuto seriamente
> che i giochi siano inutili.
> UMBERTO ECO

Domino problems have been important for proving undecidability and lower complexity bounds for subclasses of the predicate calculus [14], [11], [16], [8] and for various types of propositional logics [4], [12]. Their simple combinatorial structure allows elegant reductions to formulae with a simple syntactical structure, e.g., with a simple quantifier prefix. As has been mentioned before, the classical domino problems are existential in nature and thus provide a tool for dealing with *nondeterministic* complexity classes.

For measuring the complexity of specific (decidable) logical theories, *alternating time* complexity classes seem most appropriate: a huge amount of results is known to

the effect that the decision problem of a theory is complete in some class $\bigcup_{c>0}$ ATIME $(\exp_r(cn), cn)$ for an $r \geqq 1$. ($\exp_r$ is the r-fold iterated exponential function.) For a survey on such results see, e.g., [5].

The fact that almost all interesting mathematical theories have at least exponential complexity, together with the observation that decision problems occurring in mathematical practice are usually formulated by formulae with quite a simple quantifier prefix, leads to the question, how the complexity of a formula in a theory Th depends on its quantifier structure: is it possible to find subclasses of Th which are strong enough to express interesting statements but which are much easier to decide than the whole theory. It has turned out that a restriction of the number of quantifier alternations usually results in an exponential decrease of complexity (see [9] for a survey on such results). For instance, the theory of real numbers with addition RA is complete in $\bigcup_{c>0}$ ATIME $(2^{cn}, cn)$ [2], whereas its subclasses RA_m of sentences with at most m quantifier alternations are contained in the mth level of the polynomial-time hierarchy, as was shown by Sontag [20]. To be precise RA_m is the disjoint union of $RA_{m,\exists}$ and $RA_{m,\forall}$ (consisting of those sentences in RA_m which begin with an $\exists$, respectively, with an $\forall$) and

$$RA_{m,\exists} \text{ is } \Sigma_m^p\text{-complete},$$

$$RA_{m,\forall} \text{ is } \Pi_m^p\text{-complete}.$$

Using domino games we are able to show that this does not hold for all theories. A counterexample is provided by the theory of Boolean algebras BA. Its decision problem is treated by Kozen [15] who shows that BA is complete in $\bigcup_{c>0}$ ATIME $(2^{cn}, n)$, i.e., it has essentially the same complexity as RA. This computational equivalence does not, however, extend to the subclasses with bounded quantifier alternations: in the case of Boolean algebras, these still have exponential complexity as shown in Theorem 8.1.

THEOREM 8.1. *There is a positive constant c such that for all $m \geqq 1$*

$$BA_{m+1} \notin \text{ATIME}(2^{cn/m}, m).$$

In particular, we already have a nondeterministic exponential lower bound for the sentences in BA with only two alternations:

$$BA_2 \notin \text{NTIME}(2^{cn}).$$

The proof proceeds by reduction from GAME $(2^n, m)$ [10]. Similar results can be obtained for other theories, e.g., Th $(\mathbf{N}, |)$ and Th $(\mathbf{N}, \perp)$, the theories of natural numbers with divisibility and with coprimeness, respectively. These results and related questions will be discussed in a subsequent paper.

What is the particular merit of domino games for obtaining such results? In order to prove lower complexity bounds for a (subclass of a) logical theory we normally perform (at least implicitly) two steps: First we define large finite sets by short formulae of the theory and then we show that these sets can be given the minimum of structure which is necessary to encode Turing machine computations. For the first step domino games certainly offer no help—reductions from domino games require the same large sets as direct encodings of the computations. (In the case of Boolean algebras this is possible because the free Boolean algebra generated by n elements has cardinality 2^{2^n}.) But the simple combinatorial structure of dominoes makes the second step easier in many cases. In particular, if we look for reductions for formulae which are not only short in length but which also have a simple quantifier prefix, then we believe that

domino tilings (for nondeterministic complexity) and domino games (for alternating complexity) may be very useful.

REFERENCES

[1] R. BERGER, *The undecidability of the domino problem*, Mem. Amer. Math. Soc., 66 (1966).

[2] L. BERMAN, *The complexity of logical theories*, Theoret. Comput. Sci., 11 (1980), pp. 71-77.

[3] A. K. CHANDRA, D. C. KOZEN, AND L. STOCKMEYER, *Alternation*, J. Assoc. Comput. Mach., 28 (1981), pp. 114-133.

[4] B. CHLEBUS, *Domino-tiling games*, J. Comput. System Sci., 32 (1986), pp. 374-392.

[5] K. J. COMPTON AND C. W. HENSON, *A simple method for proving lower bounds on the computational complexity of logical theories*, Annals Pure Appl. Logic, to appear.

[6] H.-D. EBBINGHAUS, *Undecidability of some domino connectability problems*, Z. Math. Logik Grundlag. Math., 28 (1982), pp. 331-336.

[7] ———, *Domino threads and complexity*, in Computation Theory and Logic, E. Börger, ed., Lecture Notes in Computer Science, Vol. 270, Springer-Verlag, Berlin, New York, 1987, pp. 131-142.

[8] M. FÜRER, *The computational complexity of the unconstrained limited domino problem (with implications for logical decision problems)*, in Logic and Machines. Decision Problems and Complexity, E. Börger, ed., Lecture Notes in Computer Science, Vol. 171, Springer-Verlag, Berlin, New York, 1984, pp. 312-319.

[9] E. GRÄDEL, *Dominoes and the complexity of subclasses of logical theories*, Annals Pure Appl. Logic, 43 (1989), pp. 1-30.

[10] ———, *Domino games with an application to the complexity of Boolean algebras with bounded quantifier alternations*, in Proc. STACS 88, Lecture Notes in Computer Science, Vol. 294, Springer-Verlag, Berlin, New York, 1988, pp. 98-107.

[11] Y. GUREVICH, *The decision problem for standard classes*, J. Symbolic Logic, 41 (1976), pp. 460-464.

[12] D. HAREL, *Recurring dominoes: making the highly undecidable highly understandable*, in Foundations of Computation Theory, Lecture Notes in Computer Science, Vol. 158, Springer-Verlag, Berlin, New York, 1983, pp. 177-194.

[13] N. D. JONES AND W. T. LAASER, *Complete problems for deterministic polynomial time*, Theoret. Comput. Sci., 3 (1977), pp. 105-117.

[14] A. S. KAHR, E. F. MOORE, AND H. WANG, *Entscheidungsproblem reduced to the* $\forall\exists\forall$*-case*, Proc. Nat. Acad. Sci. U.S.A., 48 (1962), pp. 365-377.

[15] D. KOZEN, *Complexity of Boolean algebras*, Theoret. Comput. Sci., 10 (1980), pp. 221-247.

[16] H. R. LEWIS, *Unsolvable Classes of Quantificational Formulas*, Addison-Wesley, Reading, MA, 1979.

[17] H. R. LEWIS, AND C. H. PAPADIMITRIOU, *Elements of the Theory of Computation*, Prentice-Hall, Englewood Cliffs, NJ, 1981.

[18] D. MYERS, *Decidability of the Tiling Connectability Problem*, Notices Amer. Math. Soc., 19 (1980), pp. A-441.

[19] M. W. P. SAVELSBERGH AND P. VAN EMDE BOAS, BOUNDED TILING, *an alternative to* SATISFIABILITY?, in Proc. 2nd Frege Conference, Schwerin 1984, G. Wechsung, ed., Akademie Verlag, Mathematische Forschung, 20, pp. 354-363.

[20] E. D. SONTAG, *Real addition and the polynomial-time hierarchy*, Inform. Process. Lett., 20 (1985), pp. 115-120.

[21] L. STOCKMEYER, *The polynomial-time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1-22.

[22] J. TORÀN, *Structural properties of the counting hierarchies*, Ph.D. thesis, Facultat d'Informática de Barcelona, Barcelona, Spain, 1988.

[23] H. WANG, *Proving theorems by pattern recognition* II, Bell System Tech. J., 40 (1961), pp. 1-41.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 805–820, October 1990

003

THE HOUGH TRANSFORM HAS $O(N)$ COMPLEXITY ON $N \times N$ MESH CONNECTED COMPUTERS*

R. E. CYPHER†, J. L. C. SANZ‡, AND L. SNYDER†

Abstract. This paper presents algorithms for implementing an important image processing operation, the Hough transform, on a mesh connected computer (MCC). The MCC consists of an $N \times N$ array of processors, each of which holds a single pixel of the image. The MCC operates in a Single Instruction Stream, Multiple Data Stream (SIMD) mode, which is in agreement with the hardware constraints found in existing meshes. Five algorithms for computing the Hough transform are presented. These algorithms use a number of different techniques, and they have varying time complexities and architectural requirements. The most notable algorithm presented computes any P angles of the Hough transform in $O(N+P)$ time and uses only a constant amount of memory per processor. Because the Hough transform is a particular case of the discrete Radon transform, all of the algorithms will be presented for computing the Radon transform of gray-level images.

Key words. radon transform, Hough transform, image processing, parallel algorithms, mesh array computers

AMS(MOS) subject classifications. 68U10, 68Q20, 68Q35, 68Q80

1. Introduction.

Parallel computing for image processing and computer vision has received considerable attention during the last few years [30], [14], [3], [10]. Technological advances have made possible new processor interconnection topologies [35], [17] and fine-grained architectures [2], [19]. Mesh connected computers (MCCs) are of particular interest to the image processing community [21], [31], [12], [8], [37], [27], [13]. Many practical algorithms have been implemented on fine-grained MCCs [29], [9] and, more recently, on coarse-grained MCCs [23]. Theoretical advances in developing algorithms for MCCs have also been reported [26], [24], although it will take some time before these ideas can be put into practice. This paper addresses the issue of calculating the Hough transform on an MCC. The fastest previously published algorithm for this problem calculates P projections of the Hough transform in $O(NP)$ time [34]. Algorithms will be presented here that calculate P such projections in $O(N+P)$ time. A different algorithm with the same asymptotic complexity was created simultaneously and independently by Guerra and Hambrusch [16].

Section 2 of this paper summarizes the basic hardware characteristics of MCCs. This material is important because it justifies the assumptions for the computing models used in § 3, where the main results of the paper are given. Conclusions are presented in § 4.

2. Mesh connected computers.

An MCC consists of N^2 processing elements (PEs) arranged in an $N \times N$ array (see Fig. 1). Each PE consists of a processor and an associated memory. The PEs operate in a Single Instruction Stream, Multiple Data Stream (SIMD) mode, with all control signals coming from a single control unit. The control unit reads instructions from its private memory, decodes them, and broadcasts the control signals to the PE array. In addition to broadcasting the control information to the processors, the control unit typically sends addresses to the memory units, so

* Received by the editors June 1, 1987; accepted for publication (in revised form) September 29, 1989.
† Computer Science Department, University of Washington, Seattle, Washington 98195. The work of the first author was supported by the National Science Foundation grant fellowship program. The work of the third author was supported in part by National Science Foundation grant DCR8416878.
‡ Computer Science Department, IBM Almaden Research Center, San Jose, California 95120.

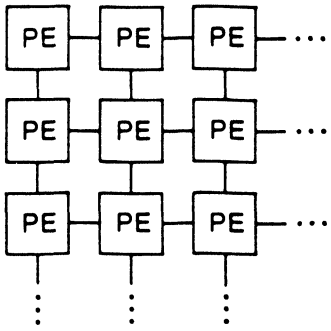


FIG. 1. *Mesh connected array.*

every PE accesses the same memory location at a given time. There is usually no direct data connection from the controller to the array. There are, however, situations in which a number must be broadcast from the controller to the PEs. This broadcast can be accomplished by using the control lines from the controller to the array and having the PEs calculate the number one bit at a time. For instance, if each PE has the ability to calculate arbitrary Boolean functions, they can be directed to calculate the function that always returns TRUE for those bit positions that contain a 1 and to calculate the function that always returns FALSE for the remaining bit positions.

It is common to provide a special register, called a mask register, in each PE. When an instruction is sent from the controller to the array of PEs, only those PEs with a 1 in their mask register perform the instruction; all others do nothing. This allows operations to be performed on a subset of the PEs in a data dependent manner. Of course, there are some instructions that operate on all PEs regardless of the setting of the mask registers, thus allowing the disabled PEs to be used again.

The array structure is easy to construct because it is regular, it has short connections, it requires only four connections per PE, and it is possible to build in two dimensions without having any connections cross. Each PE that is not on an edge of the array has a direct connection with its four nearest neighbors. The edge PEs can either be connected to a smaller number of PEs or they can be connected to the corresponding PEs on the opposite edge. In the latter case, the top row is connected to the bottom row and the leftmost column is connected to the rightmost column, so the interconnections logically form a torus. The construction of such a machine in two dimensions requires that some connections cross.

While the above description provides a simple, idealized model of MCCs, actual computers vary slightly, as is seen by examining MCCs that have been built [15], such as the CLIP4 [8], the GAPP [27], and the MPP [2].

Before presenting the architectures that will be studied in the next section, it is useful to examine the issue of SIMD versus MIMD (Multiple Input Multiple Data) control in some detail. Both of the architectures presented later in this section have a large number of PEs that operate synchronously under the direction of a single controller. The controller broadcasts instructions to the PEs which are then executed in parallel by the PEs. Because all PEs perform the same operation (such as addition or multiplication) at the same time, the machines can be considered to operate in an SIMD mode. However, there are actually three aspects to the distinction between the SIMD and MIMD modes of operation. First, the operation performed at a given time can be the same in every PE, which corresponds to an SIMD mode of operation, or different operations can be performed at a given time in different PEs, which corre-

sponds to an MIMD mode of operation. Second, the memory addresses of the operands and result can be the same for every PE at a given time, or they can be different in different PEs. The first case will be referred to as SIMD addressing, while the second will be referred to as independent addressing. It is possible to have independent addressing even when all PEs are operating under the direction of a single controller. This can be accomplished by using indirection. The third aspect of the distinction between SIMD and MIMD operation is whether or not every PE receives data from the same neighbor (e.g., the North neighbor) at a given time. The case where every PE does receive data from the same neighbor at a given time will be called SIMD communication, and the other case will be called independent communication. Independent communication can be implemented when there is a single controller, by using indirection to choose the communication port from which data will be received. All of the MCCs studied in this paper operate in an SIMD mode and use only SIMD communication.

The first architecture studied is an $N \times N$ array of PEs that processes $N \times N$ images. This will be referred to as the "plain MCC" architecture. The PEs communicate directly with their four closest neighbors, one neighbor at a time. The top and bottom rows are connected and the leftmost and rightmost columns are connected, so the interconnections logically form a torus. It is assumed that the PEs can be selectively disabled through the use of a mask register. Different types of plain MCCs will be considered, where the types vary in the amount of memory present per PE and in the ability to perform independent addressing.

The second architecture is identical to the first except that in addition to the square array of PEs, it includes a tree of PEs per row of the array. The trees of PEs are built on top of the array, so that the PEs in each row of the array form the leaf nodes of a tree. The root of each tree has an output port for the removal of results from the tree. A second controller is provided so that the PEs internal to the tree can perform one operation while the PEs in the array perform another. This architecture will be referred to as the "augmented MCC" architecture. For the algorithms presented in § 3, it is sufficient that the PEs in the nonleaf levels of the trees be adders. These trees can be shown to be useful in solving a number of important vision problems [5]. The augmented MCC architecture is similar to the mesh-of-trees architecture [25], [36].

In both the plain and augmented MCCs, the PEs in the $N \times N$ array will be indexed by a pair of integers. PE (x, y), where $0 \leqq x < N$, $0 \leqq y < N$, is located in column x and row y, with PE $(0, 0)$ being located in the lower left-hand corner of the array.

The time and memory requirements of the algorithms will be analyzed in terms of words, where each word has $O(\log N)$ bits. It is assumed that a pixel can be stored in a single word of memory. Although operations on words that are longer than one bit are not directly supported by the bit-serial machines currently in existence, they can be implemented in software as a sequence of bit-serial operations. The time analysis will be for the worst-case data.

3. Hough and Radon transforms. The Radon transform of a gray-level image is a set of projections of the image taken from different angles. Specifically, the image is integrated along line contours defined by the equation:

$$\{(x, y): x \cos(\theta) + y \sin(\theta) = \rho\},$$

where θ is the angle of the line with respect to the positive y-axis and ρ is the (signed) distance of the line from the origin. This parameterization of the line contours is used rather than the slope-intercept parameterization so that vertical lines can be represented with finite values of the parameters. It is fair to assume that $0 < \theta \leqq \pi$, because

projections are the same for θ and for $\theta + \pi$ regardless of the value of θ. The original definition of the Radon transform involves a line integral. For digitized pictures, this integral is replaced by a weighted summation. It will be assumed that each line contour is approximated by a band that is one pixel wide. All of the pixels that are centered in a given band will contribute to the value of that band. Because each band is one pixel wide, there are at most $\sqrt{2}\,N$ values of ρ for each value of θ. The parameter P will be used to denote the number of projections (values of θ) that are to be calculated. Some of the computational aspects of the discrete form of the Radon transform and its implementation in pipeline architectures have been studied previously [33].

The Hough transform is just a particular case of the Radon transform, where the input image is binary. Both transforms have many uses in computer vision [1] and image processing [18]. The Hough transform is often used to locate the edges in an image. This application of the Hough transform has been widely used in solving industrial vision problems [11], [28].

The Hough transform has been implemented on different architectures including the GAPP [34], systolic arrays [4], and pipeline architectures [32]. Also, special architectures for computing peaks of the Hough transform have been proposed [22] and built [7]. In addition, Kushner, Wu, and Rosenfeld [20] studied the problem of implementing the Hough transform on the MPP and concluded that because the projections are at various orientations, the problem "is of a form that the fixed geometry of the MPP cannot easily handle." This section contains five new algorithms for the Radon (Hough) transform on MCCs having different time complexities and resource requirements.

One way to calculate the Hough transform on a plain MCC is to rotate the columns of the image until all of the pixels that lie in the same band (line contour) for a given projection angle are in the same row. The projection is then calculated by totaling the values for each band by using horizontal shifts and adds. The first four algorithms are all based on this general approach. One algorithm based on this technique is given in [34]. The first algorithm presented here is faster by a constant factor, and the remaining algorithms are asymptotically faster.

In the algorithm descriptions that follow, two simplifying assumptions will be made. First, it is assumed that different PEs can perform a constant number of different operations at the same time. This is not possible with SIMD machines, but it is easy to simulate in constant time by selectively disabling PEs. Second, it is assumed that whenever a PE attempts to access an array location in its memory that does not exist, the PE is disabled for that operation. Again, this is easily implemented in constant time with an SIMD machine.

3.1. Algorithm 1. The first algorithm operates on a plain MCC with a constant number of words of memory per PE and SIMD addressing. The algorithm requires $O(NP)$ time to calculate P projections of the Hough transform. First, the case where $\pi/2 < \theta \leqq 3\pi/4$ will be examined. The values of θ from $\pi/2$ through $3\pi/4$ are treated in order starting with $\pi/2$ and increasing through $3\pi/4$. For each value of θ, the controller first broadcasts the values $\sin(\theta)$ and $\cos(\theta)$ to all of the PEs. Each PE (x, y) calculates $d = x\cos(\theta) + y\sin(\theta)$. The value d is the (signed) distance from the origin to the line that passes through the point (x, y) at angle θ with respect to the positive y-axis. Next, each PE calculates $\rho = \text{floor}(d)$, which is the distance from the origin to the lower edge of the one pixel wide band that passes through (x, y) at the desired angle. Then each PE calculates $v = \text{ceiling}(\rho/\sin(\theta))$, which is the smallest value on the y-axis within the band containing the point (x, y).

Each PE then creates a record containing its pixel value and the variables x, y, and v. These records are shifted vertically (cyclically, because of the connections between the top and bottom rows) until each record is in row r where $r = v \bmod N$ (see Fig. 2). For instance, if the record originally in PE (10, 3) has a v value of 1, the record is shifted down two rows. The details of how the vertical shifting is performed will be presented shortly. After this vertical shifting, each row r will have the pixels from all bands that cross the y-axis at distance v from the origin where $v = r \bmod N$.

Because $\pi/2 < \theta \leqq 3\pi/4$, there are at most two bands with different values of v on the same row r. Also, because each band is one pixel wide, it is possible that at most two pixels in the same column of the image lie within the same band. To avoid shifting one record on top of another, the vertical shifts should be performed in two stages, one for the even y-coordinates and one for the odd y-coordinates. After all of the records have been shifted vertically until they are in row $r = v \bmod N$, they are stored so that they can be used in calculating the next projection.

Then, the pixel values of the records with odd y-coordinates are added to those of the records with even y-coordinates. Next, the total of each band is calculated by shifting the pixel sums horizontally and adding them together. This process has $(\log N) - 1$ stages, where stage i, $0 \leqq i < (\log N) - 1$, consists of 2^i left shifts of the pixel sums followed by an addition of the shifted and unshifted sums. Because each row may have the pixels for two bands within it, this summation of pixels across the rows must be performed in two passes, one for each band. The totals formed in the first column form the desired projection. The next projection is calculated in the same manner, starting with the records in the PEs where they were stored after performing the vertical shifts in the calculation of the previous projection.

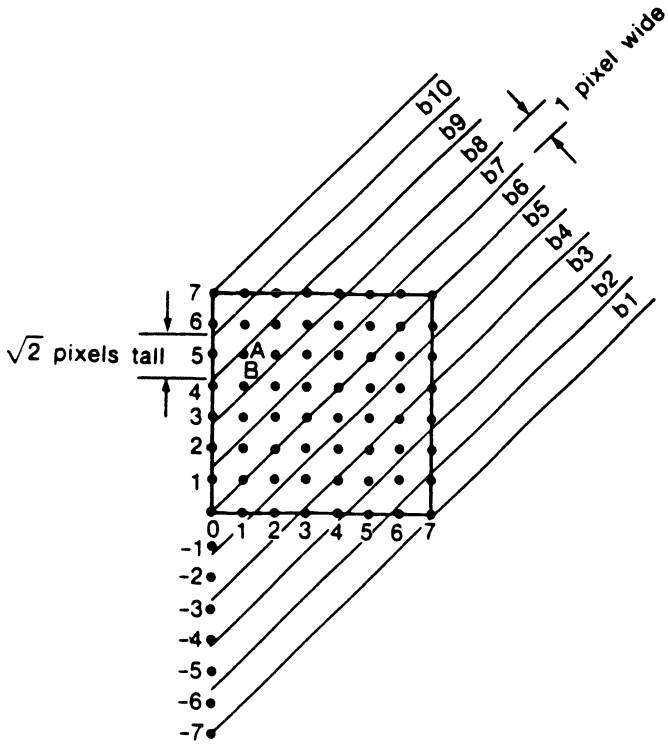


FIG. 2. *Bands for* $\theta = 3(\pi/4)$ *for an* 8×8 *image.*

By performing the projections in order and storing the records after shifting them vertically for each projection, the later projections can take advantage of the shifts performed for the earlier projections. In fact, it will be shown that only $O(N+P)$ vertical shifts are required for all of the projection angles θ where $\pi/2 < \theta \leqq 3\pi/4$. Although performing the vertical shifts in $O(N+P)$ time does not improve the asymptotic complexity of Algorithm 1, the technique will be essential for the next three algorithms. The following lemma will be useful in showing how to perform the vertical shifts efficiently.

LEMMA 1. *For all values of θ where $\pi/2 < \theta \leqq 3\pi/4$ and all values of x and y, if*

$$v = \text{ceiling (floor } (x \cos(\theta) + y \sin(\theta))/\sin(\theta))$$

then $x \cot(\theta) + y - \sqrt{2} \leqq v \leqq x \cot(\theta) + y + 1$.

Proof. First, it will be shown that $v \geqq x \cot(\theta) + y - \sqrt{2}$. From the definitions of the ceiling and floor functions and from the fact that $1 \leqq 1/\sin(\theta) \leqq \sqrt{2}$ for the given values of θ,

$$\begin{aligned} v &\geqq \text{floor } (x \cos(\theta) + y \sin(\theta))/\sin(\theta) \\ &\geqq (x \cos(\theta) + y \sin(\theta) - 1)/\sin(\theta) \\ &\geqq x \cot(\theta) + y - \sqrt{2}. \end{aligned}$$

Next, it will be shown that $v \leqq x \cot(\theta) + y + 1$. From the definitions of ceiling and floor and from the fact that $0 < \sin(\theta)$ for the given values of θ,

$$v \leqq \text{floor } (x \cos(\theta) + y \sin(\theta))/\sin(\theta) + 1 \leqq x \cot(\theta) + y + 1,$$

which completes the proof. □

If the floor and ceiling functions were not present in the calculation of v, then successively larger values of θ (in the given range) would create successively smaller values of v, and only downward shifts would be required to move each record to row $r = v \bmod N$. The floor and ceiling functions could cause a larger value of θ to have a slightly larger value of v, thus requiring upward shifts in addition to the downward shifts. However, Lemma 1 places a bound on the number of upward shifts that are required. From Lemma 1, it follows that if $\pi/2 < \theta_i < \theta_j \leqq 3\pi/4$ and if v_i and v_j are the v values for θ_i and θ_j, respectively, for any pixel (x, y), then $v_j - v_i \leqq 1 + \sqrt{2}$. Because v_i and v_j are integers, $v_j - v_i \leqq 2$. As a result, the algorithm for performing the vertical shifts for the records with even (or odd) y-coordinates and for a given value of θ consists of performing two upward shifts and then performing downward shifts until each record has reached row $r = v \bmod N$. Given this technique of implementing the vertical shifts, it will now be shown that $O(N+P)$ such shifts suffice.

LEMMA 2. *If the vertical shifts are implemented as described above, then P projections in the range $\pi/2 < \theta \leqq 3\pi/4$ require only $O(N+P)$ vertical shifts.*

Proof. First, consider the vertical shifts required for the pixels with even y-coordinates. From Lemma 1, it follows that if $\pi/2 < \theta_i < \theta_j \leqq 3\pi/4$ and if v_i and v_j are the v values for θ_i and θ_j, respectively, for any pixel (x, y), then $v_i - v_j \leqq x(\cot(\theta_i) - \cot(\theta_j)) + \sqrt{2} + 1$. Therefore, at most $N(\cot(\theta_i) - \cot(\theta_j)) + \sqrt{2} + 5$ vertical shifts are required to process the pixels with even y-coordinates for angle θ_j after angle θ_i has been processed. As a result, P projection angles in the range $\pi/2 < \theta \leqq 3\pi/4$ require at most $N(\cot(\pi/2) - \cot(3\pi/4)) + P(\sqrt{2}+5) = N + P(\sqrt{2}+5) = O(N+P)$ vertical shifts for the pixels with even y-coordinates. Because the pixels with odd y-coordinates require an equal number of vertical shifts, a total of $O(N+P)$ vertical shifts are sufficient, thus completing the proof. □

The projections where $\pi/4 < \theta \leqq \pi/2$ are calculated in the same way, starting with the original unshifted image and processing the values of θ in decreasing order. There is a problem with using the same algorithm to calculate the projections for $0 < \theta \leqq \pi/4$ and for $3\pi/4 < \theta \leqq \pi$. These projections use bands that are more vertical than horizontal, so it is possible that many bands with different values of v would have the same value of r. Because these bands would have to be totaled sequentially, the total time for the algorithm would increase. One solution is to modify the algorithm so that the use of rows and columns is reversed. Then each band would be shifted horizontally (instead of vertically) until each of the bands occupies a single column of PEs and then the bands would be totaled within columns.

However, a different technique will be presented here. This technique has the advantage of being easily modified to create an algorithm for an augmented MCC. This algorithm calculates the transpose of the image before calculating the projections where $0 < \theta \leqq \pi/4$ or $3\pi/4 < \theta \leqq \pi$. To obtain the transpose, the original image is shifted in four stages. In the first stage, the pixel in PE (x, y) is shifted down x times. In the second stage, the pixel in PE (x, y) after the first stage is shifted to the right y times. In the third stage, the pixel in PE (x, y) after the second stage is shifted down x times. Finally, in the fourth stage, the pixel in PE (x, y) after the third stage is shifted down $2y$ mod N times. Figure 3 demonstrates the operation of this transpose routine, and the following lemma establishes its correctness.

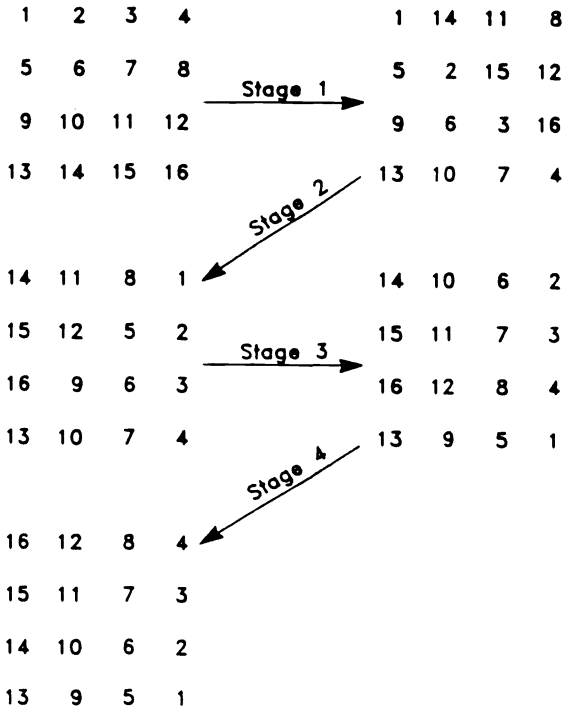


FIG. 3. *Pixel locations before and during the transpose routine.*

LEMMA 3. *The four stage process described above creates the transpose of the original image in* $O(N)$ *time.*

Proof. Each stage requires $N-1$ shifts, so a total of $4N-4=O(N)$ shifts are required. If a pixel is originally located in PE (x, y), then following stage 1 it is in PE $(x, y-x \bmod N)$, following stage 2 it is in PE $(y, y-x \bmod N)$, following stage 3 it is in PE $(y, -x \bmod N)$, and following stage 4 it is in PE (y, x), which completes the proof. □

Because the transpose is obtained in $O(N)$ time and each of the four groups of projections is calculated in $O(NP)$ time, the following theorem is proven.

THEOREM 1. *Algorithm* 1 *calculates P projections of the Hough transform in* $O(NP)$ *time on a plain* MCC *with a constant amount of memory per* PE *and* SIMD *addressing.*

3.2. Algorithm 2. In the Hough transform algorithm presented above, there are $O(N+P)$ vertical shifts and $O(NP)$ horizontal shifts. The second algorithm is identical to the first except that it uses an augmented MCC to sum the pixels within rows. The augmented MCC can calculate totals for all rows in parallel in $O(\log N)$ time. More importantly, the row totals for successive projections can be pipelined. As soon as the totals for one projection angle start up the trees of PEs, the vertical shifts for the next projection angle are started. As a result, only $O(\log N+P)$ time is spent calculating the row totals. Because $O(N+P)$ vertical shifts are performed, and because the transpose is obtained in $O(N)$ time, the entire algorithm requires $O(N+P)$ time. As was the case with the first algorithm, the second algorithm uses only a constant number of words of memory per PE and SIMD addressing. Thus, the following theorem is obtained.

THEOREM 2. *Algorithm* 2 *calculates P projections of the Hough transform in* $O(N+P)$ *time on an augmented* MCC *with a constant amount of memory per* PE *and* SIMD *addressing.*

3.3. Algorithm 3. There are two versions of Algorithm 3, depending on the relationship between the parameters N and P. The following description will be for the case $2P \leqq N$. When $2P \leqq N$, Algorithm 3 calculates P projections of a Hough transform in $O(N)$ time on a plain MCC that has $O(P)$ words of memory per PE and the ability to perform independent addressing. This algorithm also uses the technique of performing vertical shifts to place the pixels from each band into a single row of PEs, and horizontal shifts to total the pixels in each band. Algorithm 1 alternated between performing the vertical shifts for a projection and totaling the bands for that projection. Although there is some overlap between these operations in Algorithm 2, the totals for the bands of one projection start up the trees of PEs before the vertical shifts are performed for the next projection. In contrast, Algorithm 3 performs the vertical shifts for all of the projections before calculating any of the totals for the bands. First the vertical shifts for the projections where $\pi/2 < \theta \leqq 3\pi/4$ are performed in order, then the vertical shifts for the projections $\pi/4 < \theta \leqq \pi/2$ are performed. Next, the transpose of the image is calculated. Then the vertical shifts for $0 < \theta \leqq \pi/4$ and for $3\pi/4 < \theta \leqq \pi$ are performed.

Each PE contains a $2P$ element array, called the "buffer array," that is initialized to all 0s. After the vertical shifts for the ith projection angle are performed, the shifted pixels are stored in the buffer array either in location i or in location $i+P$. Specifically, the pixels that have v values (refer to Algorithm 1 for the definition of v values) where $0 \leqq v < N$ are stored in location i and the remaining pixels are stored in location $i+P$. As a result, the pixels that lie in a single band all lie in the same row of PEs and they are all at the same offset within the buffer arrays. Furthermore, no two bands are stored in the same row of PEs at the same offset. This is because, for a given projection, a row of PEs can hold pixels from at most two bands, and if pixels from two bands are

present, one of the bands is stored at offset i and the other is stored at offset $i+P$. From Lemmas 2 and 3, it follows that the above operations require $O(N+P)=O(N)$ time.

Next, the totals for the bands will be calculated in $O(N)$ time. Each PE has a variable, called band_total, that is initialized to 0. The totals for the bands are calculated by using an N stage routine. During stage k, $0 \leqq k < N$, each PE (i, j) adds location $(i-k) \bmod 2P$ from its buffer array to its band_total and then shifts its band_total 1 PE to the right. Note that this routine requires the ability to perform independent addressing. The effect of this routine is shown in Fig. 4. Once this N stage routine is completed, each band_total variable contains the total for a different band. Assuming that the leftmost column of PEs is connected to I/O ports, the band_total variables can be shifted to the left N times to remove the result from the MCC. This completes the calculation of the Hough transform in $O(N)$ time, assuming $2P \leqq N$.

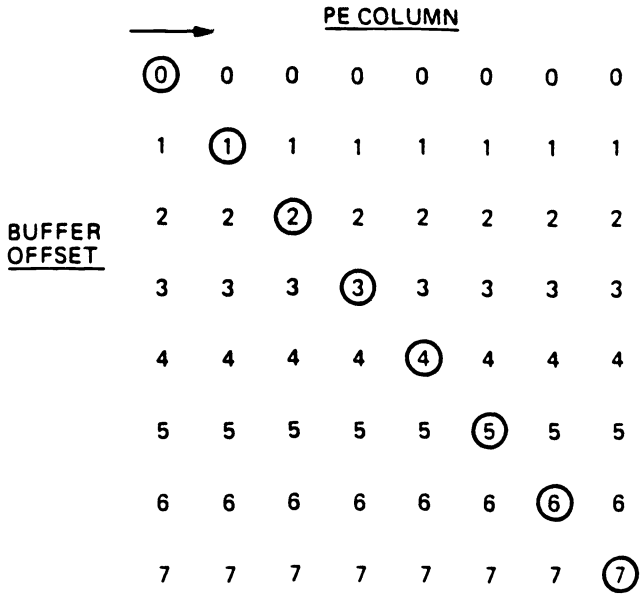


FIG. 4. *Numbers indicate band to which pixel belongs. Circled numbers are accessed first. The arrow indicates the direction in which each band_total variable moves.*

The description of Algorithm 3 for the case $2P > N$ is given next. When $2P > N$, the P projection angles are divided into ceiling $(2P/N)$ sets, each containing at most $N/2$ projection angles. Then each set of angles is processed using the algorithm for the case $2P \leqq N$. Because each set of at most $N/2$ projections requires $O(N)$ time, and because there are $O(P/N)$ such sets, the entire algorithm requires $O(P)$ time. Also, note that only $O(N)$ words of memory are required per PE. As a result, the following theorem is established.

THEOREM 3. *When $2P \leqq N$, Algorithm 3 calculates P projections of the Hough transform in $O(N)$ time on a plain* MCC *with $O(P)$ words of memory per* PE *and independent addressing. When $2P > N$, Algorithm 3 calculates P projections of the Hough transform in $O(P)$ time on a plain* MCC *with $O(N)$ words of memory per* PE *and independent addressing.*

3.4. Algorithm 4. There are two versions of Algorithm 4, depending on the relationship between the parameters N and P. The description of the algorithm for

the case $2P \leqq N$ is given next. When $2P \leqq N$, Algorithm 4 uses a plain MCC with SIMD addressing and $O(P)$ words of memory per PE to calculate the Hough transform in $O(N+P \log P)$ time. Once again, each PE has a buffer array with $2P$ entries that are initialized to 0. The same technique that was used in Algorithm 3 is used to place the vertically shifted pixels into the buffer arrays in $O(N+P)$ time. Algorithm 4 differs from Algorithm 3 in the way the bands are totaled. In Algorithm 3, independent addressing was used so that the N PEs in each row accessed different locations in their buffer arrays. In Algorithm 4, this is impossible because SIMD addressing is required. Instead, each PE (i, j) rotates the contents of its buffer array downward $i \bmod 2P$ positions. These downward rotations are cyclic, so that the contents of buffer array location j is moved to buffer array location $(i+j) \bmod 2P$. The technique used to perform these downward rotations will be explained shortly.

Once the downward rotations have been done, an N stage routine is used to calculate the totals for the bands. Each PE has a variable, called band_total, that is initialized to 0. During stage k, $0 \leqq k < N$, each PE adds location $k \bmod 2P$ from its buffer array to its band_total and then shifts its band_total one PE to the right. This procedure is illustrated in Fig. 5. Once this N stage routine is completed, each band_total variable contains the total for a different band. Assuming that the leftmost column of PEs is connected to input/output (I/O) ports, the band_total variables can be shifted to the left N times to remove the result from the MCC. This completes the calculation of the Hough transform assuming $2P \leqq N$.

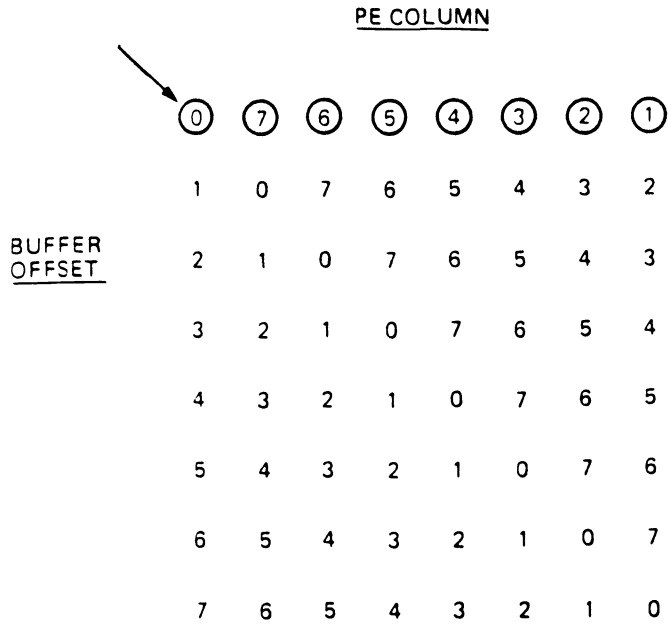


FIG. 5. *Numbers indicate band to which pixel belongs. Circled numbers are accessed first. The arrow indicates the direction in which each band_total variable moves.*

The mechanism used to perform the downward rotations of the buffer arrays was not given above. One way that these rotations could be performed would be to process in parallel only those columns of PEs that are being rotated the same amount. While processing the columns that are rotating downward x positions, $0 \leqq x < 2P$, all other columns would be disabled. Each buffer array in the enabled columns of PEs would

be copied to a temporary array in the same PE. Then the contents of this temporary array would be copied to the buffer array in the correct rotated position. Because each such rotation takes $O(P)$ time, and because $O(P)$ different rotations would be required, the rotations would require $O(P^2)$ time.

Instead of using the above technique, a routine based on the idea of a barrel shifter is used to perform the rotations in $O(P \log P)$ time. This routine consists of ceiling $(\log 2P)$ stages where stage k, $0 \leqq k <$ ceiling $(\log 2P)$, shifts the buffer arrays in the enabled PEs downward (cyclically) 2^k positions. During stage k, each PE (i, j) is enabled only if the kth bit of $(i \bmod 2P)$ is a 1, assuming that the least significant bit is the 0th bit. As a result, the buffer array in each PE (i, j) is rotated downward $i \bmod 2P$ positions. Because each stage requires $O(P)$ time, and because there are $O(\log P)$ stages, the buffer arrays are rotated in $O(P \log P)$ time. As a result, the entire algorithm requires $O(N + P \log P)$ time when $2P \leqq N$.

The algorithm for the case $2P > N$ is given next. When $2P > N$, the P projection angles are divided into ceiling $(2P/N)$ sets, each containing at most $N/2$ projection angles. Then each set of angles is processed using the algorithm for the case $2P \leqq N$. Because each set of at most $N/2$ projections requires $O(N + N \log N) = O(N \log N)$ time, and because there are $O(P/N)$ such sets, the entire algorithm requires $O(P \log N)$ time. Also, note that only $O(N)$ words of memory are required per PE. This yields the following theorem.

THEOREM 4. *When* $2P \leqq N$, *Algorithm* 4 *calculates* P *projections of the Hough transform in* $O(N + P \log P)$ *time on a plain* MCC *with* $O(P)$ *words of memory per* PE *and* SIMD *addressing. When* $2P > N$, *Algorithm* 4 *calculates* P *projections of the Hough transform in* $O(P \log N)$ *time on a plain* MCC *with* $O(N)$ *words of memory per* PE *and* SIMD *addressing.*

3.5. Algorithm 5. Finally, Algorithm 5 uses a plain MCC with SIMD addressing and only a constant number of words of memory per PE to calculate P projections of a Hough transform in $O(N + P)$ time. The technique used differs significantly from the techniques used in the four previous algorithms. The previous algorithms all performed vertical shifts on the image in order to simplify the totaling of the pixels in each band. Algorithm 5, however, does not perform such vertical shifts. Instead, the total for each band is calculated by having a variable, called "band_total," visit all of the pixels in the band. As a band_total encounters a pixel in its band, it adds the pixel's value to itself. An informal description of the algorithm is given next.

In order to understand how the algorithm works, it is useful to first examine how a single band_total variable is moved across the image. This band_total is assigned a particular value of ρ and θ. It must move across the image in such a manner that it visits all of the pixels (x, y) where floor $(x \cos(\theta) + y \sin(\theta)) = \rho$. The set of pixels that must be visited by this band_total will be referred to as the pixels "owned" by the band_total. An example of the set of pixels owned by a band_total is shown in Fig. 6. Note that because the bands are one pixel wide, a band_total owns at least one pixel in each column, except where the bands extend beyond the upper or lower boundaries of the image.

In the following discussion, it will be assumed that $\pi/2 < \theta \leqq 3\pi/4$, but the techniques that will be presented are easily adapted to the other cases. In particular, when $\pi/4 < \theta \leqq \pi/2$, the roles of up and down shifts are reversed. The remaining cases are identical except that the roles of rows and columns are reversed.

Because it is assumed that $\pi/2 < \theta \leqq 3\pi/4$, each band_total owns at most two pixels in each column. One way the band_total could visit these pixels would be by

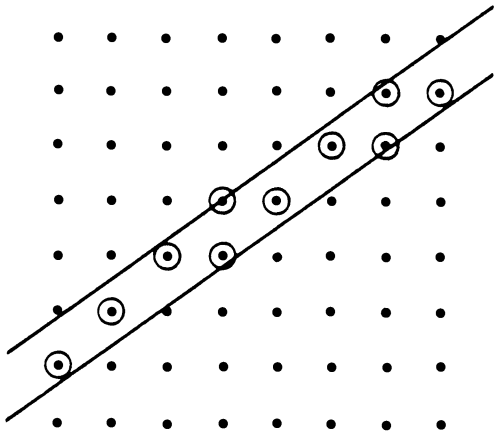


FIG. 6. *Pixels owned by a given band.*

visiting all of the PEs that contain them. This technique has the disadvantage that it requires shifting the band_total vertically in order to access both of the pixels that it owns in a single column. The algorithm presented here avoids these vertical shifts of the band_total by performing a single downward shift of the image. Thus there are two versions of the image, the original version and the down-shifted version. The band_total is moved across the image, visiting the PE that contains the lower of the band_total's pixels in each column. This PE contains both of the band_total's pixels in that column, one from the original version of the image and the other from the down-shifted version.

The set of PEs that will be visited by the band_total in Fig. 6 is shown in Fig. 7. The set of PEs that a band_total must visit will be referred to as the PEs that are "owned" by the band_total. In order for a band_total to visit all of the PEs that it owns, it is placed initially in the leftmost PE that it owns. It is then shifted one column to the right and then shifted up one row, if necessary, to visit the next PE that it owns. This process of shifting to the right and optionally shifting up is repeated until the band_total has visited all of its PEs. The decision of whether or not the band_total should be shifted upwards is based on a local calculation that will be explained below.

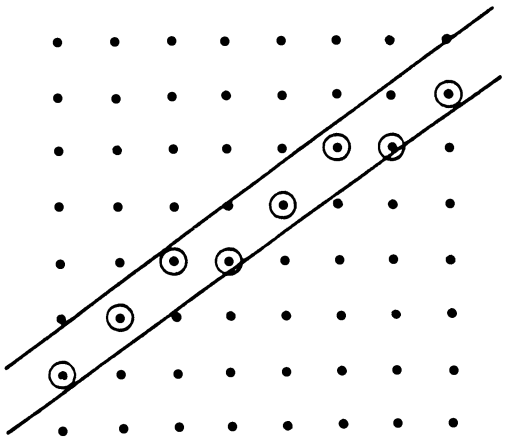


FIG. 7. PEs *owned by a given band.*

The fact that at most one upward shift is required following each shift to the right follows from the fact that $\pi/2 < \theta \leqq 3\pi/4$.

Having examined the behavior of a single band_total, it is now possible to explain how the band_totals operate in parallel. First consider the calculation of a single projection angle θ, still assuming that $\pi/2 < \theta \leqq 3\pi/4$. This projection is calculated by placing the band_totals for this value of θ in the first column of PEs. Each band_total is placed in the PE that it owns in the first column (for the time being, ignore band_totals for bands that do not own a PE in both the first and last columns). The PE that is owned by a given band_total sets its column_contrib (column contribution) variable to the sum of the pixels that are owned by the given band_total in the given column. The band_total adds this column_contrib to its current value. The band_totals are then shifted right to the second column and some of them are shifted up so that all of them are placed in the PE that they own in the second column. Again, the PEs set their column_contrib variables and these variables are added to the band_totals. This procedure is repeated until all of the columns have been visited. At any one time, the band_totals for the given projection angle θ are all located in the same column of PEs. Note that collisions between band_totals are impossible because the paths of two band_totals with the same value of θ can never cross.

In order to calculate the projection for a given angle θ, it is therefore necessary to visit the PEs in the first column, then the PEs in the second column, etc., until all of the columns of PEs have been visited. Thus, the PEs that are used in calculating one projection of the Hough transform form a wave that propagates from the left to the right. As a result, it is possible to pipeline the calculation of the Hough transform by starting the calculation of the second projection angle as soon as the first projection has finished using the PEs in the first column. As many as N projections can be pipelined in this manner at one time. Because the calculation of a single projection requires $O(N)$ time, and because the calculation of P projections can be pipelined, the calculation of the entire Hough transform can be performed in $O(N+P)$ time.

The above discussion ignored a number of details of the algorithm. In particular, it did not specify how the band_totals know when to shift up, how the column_contrib variables are calculated, or how bands that do not include pixels in both the first and last columns are handled. These issues are addressed next. When the calculation of a new projection angle θ is started, the values $\cos(\theta)$ and $\sin(\theta)$ are broadcast from the controller to the PEs in the first column. The values $\cos(\theta)$ and $\sin(\theta)$ are then shifted to the right whenever the corresponding band_totals are shifted to the right. In addition, each band_total is accompanied by a band_number that identifies which band it is following. After performing a right shift of the band_total, band_number, $\sin(\theta)$, and $\cos(\theta)$, the PEs calculate $d = x\cos(\theta) + y\sin(\theta)$ and $\rho = \text{floor}(d)$.

Then each PE determines if it is the lowest PE in its column with its value of ρ. This is done by shifting the ρ values up one row. Each PE compares the received ρ value with its own ρ value, and if they match, sets its own ρ value to infinity. At this point, the PEs with finite ρ values are the ones that are owned by some band_total for the current projection angle θ. Next, the (possibly infinite) ρ values are shifted down one row. The down-shifted ρ values are used to determine whether or not there are two pixels in the given column that lie in the same band. If a PE's own ρ value is finite and the ρ value that it received from the PE above it is infinite, then the PE knows that it contains the lower of two pixels that lie in the same band. If a PE's own ρ value is finite and the ρ value that it received from the PE above is finite, then the PE knows that it contains the only pixel in its column that lies in its band. Based on this information, each PE with a finite ρ value sets its column_contrib variable to the

sum of the pixels in its column and band. Next, each band_total examines the ρ value of the PE in which it is located. If the PE has a finite ρ value that matches the band_number, then the band_total is in the correct PE. Otherwise, the band_total and band_number are shifted up one row. The column_contrib is then added to the band_total.

So far, band_totals have only been created for bands that contain pixels in the leftmost column. However, some bands may not contain any pixels in the leftmost column. Such a band is handled in a similar manner, with the only difference being that its band_total variable is created in the leftmost column that has a pixel in the band. Also, it has been assumed that all of the bands continue all the way to the rightmost column, although in reality some do not (they go off the top or bottom of the image). When a band_total goes off the top of the image before reaching the right-hand edge, it arrives in the bottom row of PEs because of the toroidal connections between the top and bottom rows. It continues to follow the wave of processing associated with its projection angle θ, and it continues to follow its band, but it no longer is increased by the column_contrib variables that it encounters. The description given above was a simplification because it assumed that each PE has a single band_total variable at a given time. In reality, a PE can have two band_total variables at once: one that is actively visiting PEs in its band and one that has completed its calculations and has gone off the top or bottom of the image. This completes the description of Algorithm 5. The following theorem has been established.

THEOREM 5. *Algorithm 5 calculates P projections of the Hough transform in $O(N+P)$ time on a plain* MCC *with a constant number of words of memory per* PE *and* SIMD *addressing.*

A detailed analysis of the algorithm shows that it requires $20P+48N+4$ communication operations, and a similar number of local operations, to calculate P projections [6]. While this is fairly fast, it is possible to improve the speed in certain situations. If the same set of projection angles is used for calculating the Hough transform for a large number of images, the values of $\sin(\theta)$ and $\cos(\theta)$ do not have to be shifted across the image. This is because the path of each band_total variable across the image is completely determined by the values of θ chosen for the Hough transform. Instead of shifting the values of $\sin(\theta)$ and $\cos(\theta)$ across the image to calculate the paths of the band_totals, the paths can be precomputed and stored in the PEs. Each PE can contain a single bit for each time a "where" clause in the program is executed. This bit indicates whether or not the "where" clause is true at the given time. This requires an additional P bytes of memory per PE. When this approach is used, only $6P+14N+4$ communication operations are required to calculate P projections [6]. In addition, this approach requires fewer local operations and it requires no multiplications. Finally, it should be noted that in any case the multiplications can be avoided if the values of $\cos(\theta)$ and $\sin(\theta)$ are sufficiently accurate. The function $x\cos(\theta)+y\sin(\theta)$ can then be calculated by using forward differencing.

4. Conclusion. Five new algorithms for calculating the Hough transform on MCCs have been presented. The asymptotically fastest algorithm previously published requires $O(NP)$ time to calculate P projections of the Hough transform, whereas a number of the algorithms presented here require only $O(N+P)$ time. In addition to the theoretical value of these algorithms, the authors hope that they will prove fast enough to have practical applications. Typical values of the parameters N and P are 512 and 180, respectively, so the algorithms presented here do seem to offer a time savings for realistic problems.

Acknowledgment. The authors thank the anonymous reviewer for his many helpful comments.

REFERENCES

[1] D. BALLARD AND C. BROWN, *Computer Vision*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[2] K. E. BATCHER, *Design of a massively parallel processor*, IEEE Trans. Comput., C-29 (1980), pp. 836-840.
[3] V. CANTONI AND S. LEVIALDI, *Pyramidal Systems for Computer Vision*, NATO ASI Series, Computer and Systems Science, Vol. 25, Springer-Verlag, Berlin, New York, 1987.
[4] H. CHUANG AND C. LI, *A systolic array for straigl t line detection by modified Hough transform*, IEEE Workshop on Computer Architecture for Patter1 Analysis and Image Database Management, 1985, pp. 300-304.
[5] R. CYPHER AND J. L. C. SANZ, *Parallel algorithms and architectures for image processing and computer vision*, in preparation.
[6] R. CYPHER, J. L. C. SANZ, AND L. SNYDER, *The Hough transform has $O(N)$ complexity on* SIMD *$N \times N$ mesh array architectures*, Tech. Report 87-07-01, Dept. of Computer Science, University of Washington, Seattle, WA, July 1987.
[7] J. DITURI, F. M. RHODES, AND D. DUDGEON, *Hough transform system*, 1986 Workshop on VLSI Signal Processing, Los Angeles, CA, November 1986.
[8] M. J. B. DUFF, *Review of the* CLIP *image processing system*, National Computer Conference, Anaheim, CA, 1978.
[9] ———, *Real Applications on* CLIP4, in Integrated Technology for Parallel Image Processing, S. Levialdi, ed., Academic Press, London, 1985.
[10] ———, ED., *Intermediate-Level Image Processing*, Academic Press, L ɔndon, 1986.
[11] C. DYER, *Gauge inspection using Hough transforms*, IEEE Trans. PA1I, PAMI-5 (1983), pp. 621-622.
[12] C. R. DYER AND A. ROSENFELD, *Parallel image processing by memory-augmented cellular automata*, IEEE Trans. PAMI, PAMI-3 (1981), pp. 29-41.
[13] T. J. FOUNTAIN, *Plans for the* CLIP7 *chip*, Integrated Technology for Parallel Image Processing, S. Levialdi, ed., Academic Press, New York, 1985, pp. 199-214.
[14] ———, *Array architectures for iconic and symbolic image processing*, 8th Internat. Conference on Pattern Recognition, 1986, pp. 24-33.
[15] F. A. GERRITSEN, *A comparison of the* CLIP4, DAP *and* MPP *processor-array implementations*, in Computing Structures for Image Processing, M. Duff, ed., Academic Press, London, 1983, pp. 2-23.
[16] C. GUERRA AND S. HAMBRUSCH, *Parallel algorithms for line-detection on a mesh*, IEEE Workshop on Computer Architecture for Pattern Analysis and Machine Intelligence, 1987, pp. 99-106.
[17] D. HILLIS, *The Connection Machine*, MIT Press, Cambridge, MA, 1985.
[18] E. B. HINKLE, J. L. C. SANZ, A. K. JAIN, AND D. PETKOVIC, PPPE: *New life for projection-based image processing*, J. Parallel and Distributing Comput., 4 (1987), pp. 45-78.
[19] H. IBRAHIM, J. KENDER, AND D. ELLIOT SHAW, *On the application of massively parallel* SIMD *tree machines to certain intermediate-level vision tasks*, CVGIP 36, 1986, pp. 53-75.
[20] T. KUSHNER, A. WU, AND A. ROSENFELD, *Image processing on the* MPP, Pattern Recognition, 15 (1982), pp. 121-130.
[21] S. LEVIALDI, *On shrinking binary picture patterns*, Comm. ACM, 15 (1972), pp. 7-10.
[22] H. LI, M. LAVIN, AND R. LEMASTER, *Fast Hough transform: A hierarchical approach*, CVGIP 36, January 1987, pp. 139-161.
[23] K. N. MATTHEWS, *The* CLIP7 *image analyser—A multi-bit processor array*, Ph.D. thesis, University of London, 1986.
[24] R. MILLER AND Q. STOUT, *Geometric algorithms for digitized pictures on a mesh-connected computer*, IEEE Trans. PAMI, PAMI-7 (1985), pp. 216-228.
[25] ———, *Efficient parallel convex hull algorithms*, IEEE Trans. Comput., 37 (1988), pp. 1605-1618.
[26] D. NASSIMI AND S. SAHNI, *Finding connected components and connected ones on a mesh-connected parallel computer*, SIAM J. Comput., 9 (1980), pp. 744-757.
[27] NCR Microelectronics Division, Product Description ncr45cg72, NCR Corporation, Dayton, OH, 1984.
[28] D. PETKOVIC, J. L. C. SANZ, K. MOHIUDDIN, et al., *An experimental system for disk head inspection*, Proc. of ICPR, Paris, France, October 27-31, 1986.
[29] J. L. POTTER, *Image processing on the massively parallel processor*, IEEE Trans. Comput., 1983, pp. 62-67.
[30] A. P. REEVES, SURVEY: *Parallel computer architectures for image processing*, in Computer Vision, Graphics, and Image Processing, 25 (1984), pp. 68-88.

[31] A. ROSENFELD, *Parallel image processing using cellular arrays*, IEEE Trans. Comput., 1983, pp. 14–20.

[32] J. L. C. SANZ AND I. DINSTEIN, *Projection-based geometrical feature computation for computer vision: Algorithms in pipeline architectures*, IEEE Trans. Pattern Analysis and Machine Intelligence, January 1987.

[33] J. L. C. SANZ AND E. B. HINKLE, *Computing projections of digital images in image processing pipeline architectures*, IEEE Trans. ASSP, ASSP-35 (1987), pp. 198–207.

[34] T. SILBERBERG, *The Hough transform in the geometric arithmetic parallel processor*, IEEE Workshop on Computer Architecture and Image Database Management, 1985, pp. 387–391.

[35] S. TANIMOTO, T. LIGOCKI, AND R. LING, *A Prototype Pyramid Machine for Hierarchical Cellular Logic*, L. Uhr, ed., Parallel Hierarchical Computer Vision, Academic Press, Orlando, FL, to appear.

[36] J. ULLMAN, *Computational Aspects of* VLSI, Computer Science Press, Rockville, MD, 1984.

[37] S. WILSON, *The Pixie-5000: A systolic arr ıy processor*, in Proc. IEEE Computer Society Workshop on Computer Architecture for Pattern Analysis and Image Database Management, Miami, FL, November 18–20, 1985.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 821-832, October 1990

004

AN ANALYSIS OF RANDOM d-DIMENSIONAL QUAD TREES*

LUC DEVROYE† AND LOUISE LAFOREST‡

Abstract. It is shown that the depth of the last node inserted in a random quad tree constructed from independent uniform $[0, 1]^d$ random vectors is in probability asymptotic to $(2/d)\log n$, where log denotes the natural logarithm. In addition, for $d = 2$, exact values are obtained for all the moments of the depth of the last node.

Key words. average time analysis, probability ineq' alities, random quad tree, multidimensional data structures, search tree, expected behavior, analysis of a ;orithms

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

C.R. categories. 3.74, 5.25, 5.5

1. Introduction.

Various data structures have been proposed for retrieval on composite keys (or associative retrieval) such as $k-d$ trees (Bentley (1975)), multidimensional trees (Rivest (1974); Orenstein (1982)); grids with variable-sized cells (Tamminen (1981), (1982)); quad trees (Finkel and Bentley (1974)); $k-d-b$ trees (Robinson (1981)); quintary trees (Lee and Wong (1981)); and multipaging structures (Merrett and Otoo (1981)). A partial survey of these structures can be found in Tamminen (1981) or Gonnet (1984). In this paper, we analyze random quad trees. These trees have been used with a great deal of success in computer graphics (see Woodwark (1982) and the references found there) and image processing (Hunter and Steiglitz (1979)). Detailed discussions of some common operations on quad trees, and possible improvements, can be found in Bentley, Stanat, and Williams (1977), and Samet (1980). See also the survey article by Samet (1984). The quad trees considered here are known as point quad trees since they are used to store points. Many applications require region quad trees for storing screenfuls of pixels. Random region quad trees were analyzed for example by Puech and Yahia (1985).

A **quad tree** is constructed as a binary search tree. When a key X_i occupies a node, it partitions the rectangle it belongs to orthogonally into 2^d parts (called quadrants), and thus creates 2^d new rectangles, each having X_i as a vertex. We should note here that the traversal of one node requires d comparisons. A **random quad tree** is constructed by inserting $X_1, \cdots, X_n$, independently and identically distributed uniform $[0, 1]^d$ random vectors, in the standard manner into an initially empty quad tree. We will look at D_n, the **depth** of X_n after it is inserted into the tree, where, by convention, the depth of the root is zero. The level L_n of a node is equal to its depth plus one. Other important quantities are the **average depth** $A_n = (1/n)\sum_{i=1}^n D_i$ and the **height** $\max_{1 \leqq i \leqq n} D_i$. The height is in probability asymptotic to $(c/d)\log n$, where $c = 4.31107\cdots$ is the unique solution greater than two of the equation $c \log (2e/c) = 1$ (Devroye (1987)). However, unsuccessful search times are in most cases appropriately

* Received by the editors April 27, 1988; accepted for publication (in revised form) September 7, 1989. This research was sponsored by Natural Sciences and Engineering Research Council of Canada grant A3456 and by FCAR grant EQ-1678.

† School of Computer Science, McGill University, 805 Sherbrooke Street West, Montréal, Canada H3A 2K6.

‡ Département de Mathématique et d'Informatique, Université du Québec à Montréal, Montréal, Québec Canada H3C 3P8.

measured by D_n, the depth of the last node added to the tree. Our main result is the following.

THEOREM M1. *$D_n/\log n$ tends in probability to $2/d$ as $n \to \infty$. Also, $\mathbf{E}D_n \sim \mathbf{E}A_n \sim (2/d)\log n$ as $n \to \infty$.*

For $d = 1$, the quad tree reduces to the binary search tree, and the random quad tree coincides with a binary search tree constructed from a random equiprobable permutation. Its properties, including the law of large numbers given in Theorem M1, have been obtained in a series of papers by Lynch (1965), Knuth (1973), Robson (1979), Sedgewick (1983), Pittel (1984), Mahmoud and Pittel (1984), Brown and Shubert (1984), Louchard (1987), and Devroye (1986), (1987), (1988).

In § 3, we will derive large deviation inequalities for D_n. In effect, we will prove the following theorem.

THEOREM M2. *For every $\delta > 0$, there exist positive constants a, b such that*

$$\mathbf{P}\left(\left|\frac{D_n}{(2/d)\log n} - 1\right| > \delta\right) \leqq an^{-b}.$$

The extra material needed to prove this is presented in § 2. In § 4, for the planar case ($d = 2$), we obtain exact values for $\mathbf{E}D_n$ and $\mathrm{Var}(D_n)$. Both are of the order of $\log n$. Chebyshev's inequality then gives

$$\mathbf{P}\left(\left|\frac{D_n}{(2/d)\log n} - 1\right| > \delta\right) \leqq \frac{a}{\log n},$$

which is weaker than the bound obtained in Theorem M2. The exact expressions are obtained by solving some recurrences. It should be noted that the mean was obtained independently by Flajolet et al. (1988), based upon an analysis that involves computing the generating function.

THEOREM M3. *Assume that $d = 2$. For $n \geqq 2$,*

$$\mathbf{E}D_n = H_n - \frac{1}{6} - \frac{2}{3n}$$

and

$$\mathrm{Var}(D_n) = H_n^{(2)} + \frac{1}{2}H_n + \frac{5}{9n} - \frac{4}{9n^2} - \frac{13}{6},$$

where

$$H_n \triangleq \sum_{i=1}^{n} 1/i \qquad \text{and} \qquad H_n^{(2)} \triangleq \sum_{i=1}^{n} (1/i^2).$$

In § 2, we obtain auxiliary results that allow us to prove Theorems M1 and M2. This will be done by reducing the d-dimensional problem to d one-dimensional problems for which we have ready solutions at hand. We consider the quad tree formed by consecutive insertions of $X_1, \cdots, X_{n+1}$, independently and identically distributed uniform $[0, 1]^d$ random vectors. The depth D_{n+1} of X_{n+1} is equal to the number of times the rectangle in the quad tree partition containing X_{n+1} gets "cut" by $X_1, \cdots, X_n$. We start with the full rectangle $[0, 1]^d$. The process of cutting can be summarized by a sequence of random variables (T_k, Z_k), $k \geqq 0$, where $T_0 = 0$, $Z_0 = 1$. T_k is a time counter, and Z_k is the size of the rectangle containing X_{n+1} after it has been cut precisely k times. Given (T_k, Z_k), X_{n+1} and $X_1, \cdots, X_{T_k}$, it is easy to see that $T_{k+1} - T_k$ is geometric with parameter Z_k, i.e., it takes the value i with probability $Z_k(1 - Z_k)^{i-1}$ for $i \geqq 1$. Furthermore, Z_{k+1} is distributed as the size of the rectangle containing X_{n+1}

after it has been cut precisely $k+1$ times. Note that D_{n+1} is equal to the maximal k for which $T_k \leqq n$. Thus, we have

$$\mathbf{P}(D_{n+1} \geqq k) = \mathbf{P}(T_k \leqq n).$$

We will exploit this duality and offer a study of the properties of the Z_k's in § 2.

2. Auxiliary results about spacings, records, and random cuts. Consider independently and identically distributed uniform [0, 1] random variables $U_1, \cdots, U_n$, and let S_{nx} be the size of the interval to which x, a fixed number from [0, 1], belongs.

LEMMA S1. *For any* $x \in [0,1]$,

$$S_{nx} \stackrel{L}{=} \min(x, U_1, \cdots, U_n) + 1 - \max(x, U_1, \cdots, U_n),$$

where $\stackrel{L}{=}$ *denotes equality in distribution. If* $x = U$, *and* $U, U_1, \cdots, U_n$ *is an independently and identically distributed uniform* [0, 1] *sequence, then* S_{nU} *is distributed as the second smallest of* $U, U_1, \cdots, U_n$.

Proof of Lemma S1. We verify the distributional equality in three cases, according to the signs of $\min U_i - x$ and $\max U_i - x$. Consider first the case $\min U_i \leqq x$ and $\max U_i \geqq x$. Then, define

$$V_i = \begin{cases} x - U_i \text{ if } U_i < x, \\ 1 + x - U_i \text{ if } U_i \geqq x, \end{cases} \qquad 1 \leqq i \leqq n,$$

and note that

$$\begin{aligned} S_{nx} &= \left(x - \max_{i: U_i < x} U_i\right) + \left(\min_{i: U_i \geqq x} U_i - x\right) \\ &= \min_{i: U_i < x} V_i + 1 - \max_{i: U_i \geqq x} V_i \\ &= \min_i V_i + 1 - \max_i V_i. \end{aligned}$$

It is easy to verify the two other cases now. For example, if $\min U_i \geqq x$, then

$$S_{nx} = x + 1 - \max_{i: U_i \geqq x} V_i = \min(\min_i V_i, x) + 1 - \max_i V_i.$$

The second statement of the lemma follows from a property of uniform spacings (see, e.g., Pyke (1965), (1972) for a survey), which states that the sum of any k spacings is distributed as the sum of the first k spacings. □

LEMMA S2. *For* $t \in (0,1)$,

$$\mathbf{P}(S_{nU} < t) = 1 - (1+tn)(1-t)^n \leqq (tn)^2\left(\frac{1}{2} + \frac{1}{n(1-t)}\right).$$

Also,

$$\mathbf{P}(S_{nU} > t) = (1+tn)(1-t)^n \leqq (1+tn)\, e^{-tn} \leqq e^{-(tn)^2/(2(1+tn))}.$$

For $tn \geqq 1$, *the last upper bound is not greater than* $e^{-tn/4}$.

Proof of Lemma S2. Let Y be binomial $(n+1, t)$. Then, by Lemma S1,

$$\begin{aligned} \mathbf{P}(S_{nU} < t) &= \mathbf{P}(Y \geqq 2) = 1 - \mathbf{P}(Y = 0) - \mathbf{P}(Y = 1) \\ &= 1 - \binom{n+1}{0} t^0 (1-t)^{n+1} - \binom{n+1}{1} t^1 (1-t)^n \\ &= 1 - (1-t)^n (1 - t + (n+1)t) = 1 - (1-t)^n (1+tn). \end{aligned}$$

Using $\log(1+\nu) \geqq \nu - \nu^2/2$, valid for $\nu \geqq 0$, and $\log(1-\nu) \geqq -\nu/(1-\nu)$, valid for $0 \leqq \nu < 1$, we see that

$$\mathbf{P}(S_{nU} < t) \leqq 1 - \exp\left[tn - \frac{(tn)^2}{2} - \frac{tn}{1-t}\right] \leqq -tn + \frac{(tn)^2}{2} + \frac{tn}{1-t} = (tn)^2\left(\frac{1}{2} + \frac{1}{n(1-t)}\right).$$

The second part of the lemma follows from the first one and the inequality $\log(1+\nu) - \nu \leqq -\nu^2/(2(1+\nu))$, valid for $\nu \geqq 0$. □

Let $U, U_1, \cdots, U_n$ be independently and identically uniform [0, 1] random variables, and define $[V_i, U]$ and $[U, W_i]$ as the spacings nearest to U after $U, U_1, \cdots, U_i$ have been considered, with the convent on that $V_0 = 0$, $W_0 = 1$. Let N_n be the number of indices i for which $(V_i, W_i) \neq (V_{i-1}, W_{i-1})$, $1 \leqq i \leqq n$. In Devroye (1988), it is shown that N_n is distributed as the sum of n independent Bernoulli random variables Y_i, i.e., $N_n = \sum_{i=1}^n Y_i$, where $\mathbf{E}Y_i = 2/(i+1)$. We will need to know more about the properties of N_n since N_n represents the number of times the spacing containing U is "cut" as we process the U_i's. In particular, we need solid tail bounds. These can be obtained by Chernoff's exponential bounding technique (Chernoff (1952)).

LEMMA S3. *Define* $\mu = 2(H_{n+1} - 1)$. *For* $k \geqq \mu$,

$$\mathbf{P}(N_n \geqq k) \leqq \exp\left(-\frac{(k-\mu)^2}{2k}\right),$$

and for $k \leqq \mu$,

$$\mathbf{P}(N_n \leqq k) \leqq \exp\left(-\frac{(\mu-k)^2}{2\mu}\right).$$

Proof of Lemma S3. By Jensen's inequality, for arbitrary $\lambda > 0$,

$$\begin{aligned}\mathbf{P}(N_n \geqq k) &= \mathbf{P}\left(\sum_{i=1}^n Y_i \geqq k\right) \leqq \mathbf{E}\exp\left(\lambda \sum_{i=1}^n Y_i - \lambda k\right) \\ &= e^{-\lambda k}\prod_{i=1}^n \left(1 - \frac{2}{i+1} + \frac{2e^\lambda}{i+1}\right) \\ &\leqq \exp\left[-\lambda k + 2(e^\lambda - 1)(H_{n+1} - 1)\right].\end{aligned}$$

The exponent is minimal for $e^\lambda = k/(2(H_{n+1}-1))$. Resubstituting this value and using the notation $y = e^\lambda - 1 > 0$ gives the further upper bound

$$\exp\left[2(H_{n+1}-1)(y - (1+y)\log(1+y))\right] \leqq \exp\left[-\frac{2(H_{n+1}-1)y^2}{2(1+y)}\right],$$

where we used the fact that $y - (1+y)\log(1+y) \leqq -y^2/(2(1+y))$, which can be verified by using Taylor's series expansion with remainder. The last upper bound coincides with the first inequality in the statement of the lemma. To obtain the second inequality, we pick another $\lambda > 0$ and note that

$$\begin{aligned}\mathbf{P}(N_n \leqq k) &\leqq e^{\lambda k}\mathbf{E}(e^{-\lambda N_n}) = e^{\lambda k}\prod_{i=1}^n \left(1 - \frac{2}{i+1} + \frac{2e^{-\lambda}}{i+1}\right) \\ &\leqq \exp\left[\lambda k - 2(H_{n+1}-1)(1-e^{-\lambda})\right].\end{aligned}$$

The upper bound is minimal for $e^{-\lambda} = k/(2(H_{n+1}-1))$. We define $y = 1 - e^{-\lambda}$, and resubstitute these values to obtain the upper bound

$$\exp\left[-2(H_{n+1}-1)(y + (1-y)\log(1-y))\right] \leqq \exp\left[-(H_{n+1}-1)y^2\right],$$

where once again we used Taylor's series expansion with remainder. This concludes the proof of Lemma S3. □

Lemma S3 shows very clearly that N_n is close to its expected value, $2(H_{n+1}-1)$. We are almost ready to get to the main lemma about uniform cuts. Consider an infinite sequence of independently and identically distributed uniform [0, 1] random variables $U, U_1, \cdots, U_n, \cdots$, and let Z_k be the size of the spacing to which U belongs after it has been "cut" or "hit" k times by members of the sequence $U_1, U_2, \cdots$. In notation introduced above, $Z_k = W_i - V_i$ where (V_i, W_i) is the kth pair not equal to its predecessor. Interestingly, Z_k, S_{nU}, and N_n are connected via the following inclusions of events:

LEMMA S4. *Let $k>0$ and $t\in(0,1)$ be fixed. Then, for any positive integer n,*

$$[Z_k < t] \subseteq [S_{nU} < t] \cup [N_n < k],$$

and

$$[Z_k \geqq t] \subseteq [S_{nU} \geqq t] \cup [N_n \geqq k].$$

Proof of Lemma S4. The proof is obvious. □

We can now announce our main lemma for the **uniform k-cuts Z_k**.

LEMMA S5. *For $k\geqq 3$ and $\delta>0$, we have*

$$\mathbf{P}\left(Z_k < \exp\left[-\frac{k-1}{2}(1+2\delta)\right]\right) \leqq 6\exp[-\delta(k-1)] + \exp\left[-\frac{\delta^2(k-1)}{2(1+\delta)}\right].$$

Also, if $\delta\in(0,\frac{1}{2})$, $\delta\geqq 3/k$, and $k\geqq 2/(1-\delta)$, we have

$$\mathbf{P}\left(Z_k \geqq \exp\left[-\frac{k}{2}(1-2\delta)\right]\right) \leqq \exp\left[\frac{1}{2} - \left(\frac{1}{4}\right)e^{k\delta/2}\right] + (2e)^{\delta^2/(1-\delta)}\exp\left[-\frac{k\delta^2}{2}\right].$$

Proof of Lemma S5. From Lemmas S2, S3, and S4 we recall that for some n to be picked further on,

$$\begin{aligned}\mathbf{P}(Z_k < t) &\leqq \mathbf{P}(S_{nU} < t) + \mathbf{P}(N_n < k)\\ &\leqq (tn)^2\left(\frac{1}{2} + \frac{1}{n(1-t)}\right) + \exp\left[-(H_{n+1}-1)\left(1-\frac{k-1}{2(H_{n+1}-1)}\right)^2\right],\end{aligned}$$

valid for $k-1\leqq 2(H_{n+1}-1)$. Consider a constant $\delta\in(0,\frac{1}{2})$, and define

$$n = \left\lfloor 2\exp\left[\frac{k-1}{2}(1+\delta)\right]\right\rfloor.$$

We note that

$$H_{n+1}-1 = \sum_{i=2}^{n+1}\frac{1}{i} \geqq \int_2^{n+2}\frac{1}{x}\,dx = \log\left(\frac{n+2}{2}\right) \geqq \frac{k-1}{2}(1+\delta).$$

This implies that $k-1\leqq 2(H_{n+1}-1)$, as required. Using the fact that the function $(y-a)^2/y$ is increasing for $y>a\geqq 0$, that $n\geqq 2$ (by definition), and that $t\leqq\frac{1}{2}$ (by assumption), we see that

$$\begin{aligned}\mathbf{P}(Z_k < t) &\leqq 4t^2\exp[(k-1)(1+\delta)]\left(\frac{1}{2} + \frac{1}{n(1-t)}\right) + \exp\left[-\frac{\delta^2(k-1)}{2(1+\delta)}\right]\\ &\leqq 6t^2\exp[(k-1)(1+\delta)] + \exp\left[-\frac{\delta^2(k-1)}{2(1+\delta)}\right].\end{aligned}$$

We obtain the first half of the lemma by setting $t = \exp[((k-1)/2)(1+2\delta)]$. The condition $t \leqq 1/2$ is fulfilled when $(k-1)(1+2\delta) \geqq 2$, which is certainly the case whenever $k \geqq 3$.

Consider now the second half of the lemma. Assume that n is such that $k \geqq 2(H_{n+1}-1)$. Assume that $tn \geqq 1$. From Lemmas S2, S3, and S4, we retain that

$$\mathbf{P}(Z_k > t) \leqq \mathbf{P}(S_{nU} > t) + \mathbf{P}(N_n \geqq k) \leqq e^{-tn/4} + \exp\left[-\frac{(k-2(H_{n+1}-1))^2}{2k}\right].$$

We now choose $\delta \in (0, 1/2)$, and define

$$n = \lfloor e^{(k/2)(1-\delta)} \rfloor - 1.$$

This value of n is at least one if $k(1-\delta) \geqq 2$. It is easy to verify that $H_{n+1} - 1 \leqq k(1-\delta)/2$ so that the condition relating n and k is indeed satisfied. If we set $y = k/(2(H_{n+1}-1))$ (which, as we have seen, is at least equal to $1/(1-\delta)$), then the inequality reads

$$\begin{aligned}\mathbf{P}(Z_k > t) &\leqq e^{-tn/4} + \exp\left[-(H_{n+1}-1)\frac{(y-1)^2}{y}\right] \leqq e^{-tn/4} + \exp\left[-(H_{n+1}-1)\frac{\delta^2}{1-\delta}\right] \\ &\leqq e^{-tn/4} + (2e)^{\delta^2/(1-\delta)}\, e^{-k\delta^2/2},\end{aligned}$$

where we noted that

$$H_{n+1} - 1 \geqq \log\frac{n+2}{2} - 1 \geqq \log(e^{k(1-\delta)/2}) - \log(2e) = \frac{k(1-\delta)}{2} - \log(2e).$$

For $t = e^{-(k/2)(1-2\delta)}$, we have $tn \geqq e^{k\delta/2} - 2$, so that the upper bound becomes

$$\mathbf{P}(Z_k > t) \leqq e^{1/2-(1/4)e^{k\delta/2}} + (2e)^{\delta^2/(1-\delta)}\, e^{-k\delta^2/2}.$$

Also, the condition $tn \geqq 1$ is fulfilled if $k\delta \geqq 3$. □

3. A law of large numbers for quad trees. The purpose of this section is to prove Theorem M1.

THEOREM M1. *$D_n/\log n$ tends in probability to $2/d$ as $n \to \infty$. Also, $\mathbf{E}D_n \sim \mathbf{E}A_n \sim (2/d)\log n$ as $n \to \infty$.*

Recall the definition of T_k and Z_k from the Introduction. We have

$$\mathbf{P}(D_{n+1} \geqq k) = \mathbf{P}(T_k \leqq n) \leqq \mathbf{P}(T_k - T_{k-1} \leqq n).$$

Observe that $Z_k = \prod_{i=1}^{d} Z_k(i)$, where the $Z_k(i)$'s are independently and identically distributed random variables distributed as the uniform k-cut dealt with in Lemma S5. We choose a small positive constant δ, and define $q = \exp(-(k-1)(1-2\delta)/2)$. Let A be the event that $\max_i Z_{k-1}(i) \leqq q$. By an obvious left tail bound for the geometric distribution and Lemma S5, we have

$$\begin{aligned}\mathbf{P}(D_{n+1} \geqq k) &\leqq \mathbf{P}(A,\, T_k - T_{k-1} \leqq n) + \mathbf{P}(A^c) \\ &\leqq nq^d + d \times \mathbf{P}(Z_{k-1}(1) > q) \qquad (1) \\ &\leqq nq^d + d(e^{1/2-(1/4)e^{(k-1)\delta/2}} + (2e)^{\delta^2/(1-\delta)}\, e^{(k-1)\delta^2/2})\end{aligned}$$

if $\delta \geqq 3/(k-1)$ and $k - 1 \geqq 2/(1-\delta)$. As $k \to \infty$, the upper bound is $nq^d + o(1)$. If we now take

$$k - 1 = \left\lfloor \frac{2}{d(1-3\delta)} \log n \right\rfloor,$$

then it is easy to verify that $nq^d = o(1)$ as well. Hence, $\mathbf{P}(D_{n+1} \geqq k) = o(1)$, proving one half of the theorem (since δ is arbitrary). In fact, $\mathbf{P}(D_{n+1} \geqq k) = O(\log^{-R} n)$ for any positive constant R.

The second half is proved similarly. Because we obtained exponential inequalities in the lemmas of the previous section, we can actually get away with a very crude bounding technique. Let A be the event $\min_i Z_{k-1}(i) \geqq q$ where $q = \exp(((k-2)/2)(1+2\delta))$, let $k \geqq 3$, and assume that $\delta > 0$ is an arbitrary but small constant. Then,

$$\begin{aligned}
\mathbf{P}(D_{n+1} < k) = \mathbf{P}(T_k > n) &\leqq \mathbf{P}\left(\bigcup_{j=1}^{k} \left[T_j - T_{j-1} > \frac{n}{k} \right] \right) \\
&\leqq k\mathbf{P}\left(T_k - T_{k-1} > \frac{n}{k} \right) \\
&\leqq k\left(\mathbf{P}\left(A, T_k - T_{k-1} > \frac{n}{k} \right) + \mathbf{P}(A^c) \right) \\
&\leqq k\left(\sum_{i > n/k} q^d (1-q^d)^{i-1} + d \times \mathbf{P}(Z_{k-1}(1) < q) \right) \\
&\leqq k(1-q^d)^{(n/k)-1} + kd \times \mathbf{P}(Z_{k-1}(1) < q) \\
&\leqq k \exp\left[-\left(\frac{n}{k} - 1 \right) q^d \right] + o(1)
\end{aligned}$$

whenever $k \to \infty$ (by Lemma S5 and our choice of q). If we take

$$k - 2 = \left\lfloor \frac{2}{d(1+3\delta)} \log n \right\rfloor,$$

it is a simple exercise to verify that $nq^d/k \to \infty$, which then shows that $\mathbf{P}(D_{n+1} < k) = o(1)$, as required. This concludes the proof of the weak convergence of D_n. This trivially implies that

$$\liminf_{n \to \infty} \frac{\mathbf{E}D_n}{\log n} \geqq \frac{2}{d}.$$

Also, for small $\delta > 0$, and arbitrary $M > 1$,

$$\begin{aligned}
\mathbf{E}D_n &= \int_0^n \mathbf{P}(D_n > t)\, dt \\
&\leqq 1 + \frac{2}{d(1-3\delta)} \log n + M \log n\, \mathbf{P}\left(D_n > 1 + \left\lfloor \frac{2}{d(1-3\delta)} \right\rfloor \log n \right) \\
&\quad + n\mathbf{P}(D_n > M \log n) \\
&= 1 + \frac{2}{d(1-3\delta)} \log n + o(1) + n\mathbf{P}(D_n > M \log n)
\end{aligned}$$

by the bounds obtained above. To conclude that $\mathbf{E}D_n \sim (2/d) \log n$, we need only establish that $\mathbf{P}(D_n > M \log n) = o(1/n)$ for some constant M. This follows by noting that the bound (1) with q as chosen there and $k = \lfloor M \log n \rfloor$ is $o(1/n)$ whenever $M > \max(2/\delta^2, 4/(d(1-2\delta)))$.

The statement about $\mathbf{E}A_n$ finally follows easily from the statement regarding $\mathbf{E}D_n$. □

4. Some recurrences related to quad trees. In this section, we only consider the case $d=2$. As above, we let D_n be the depth of the nth node in a tree of n nodes. We also define

$$p_{n,l} \triangleq \mathbf{P}(D_n = l),$$

and note that by convention $p_{1,0}=1$, i.e., the root node is at depth zero. We begin with the following recursion:

LEMMA R1. *Let N_j, $1 \leqq j \leqq 4$, be the cardinalities of the four subtrees of the root of a random quad tree in the plane. Then*

$$\mathbf{P}(N_j = i) = \frac{H_n - H_i}{n}, \qquad 0 \leqq i \leqq n-1,$$

when $n \geqq 2$. Also, for $n \geqq 2$,

$$p_{n,l} = \frac{4}{n(n-1)} \sum_{i=0}^{n-1} i(H_n - H_i) p_{i,l-1},$$

where $p_{i,l-1}=0$ for $0 \leqq i < l$.

Proof of Lemma R1. We note that $\mathbf{P}(N_1+N_2=i)=1/n$ for $0 \leqq i < n$. Given N_1+N_2, N_1 is again uniformly distributed on $0, \cdots, N_1+N_2$. Thus,

$$\mathbf{P}(N_1=i) = \sum_{j=i}^{n-1} \frac{1}{j+1} \mathbf{P}(N_1+N_2=j) = \frac{1}{n} \sum_{j=i}^{n-1} \frac{1}{j+1} = \frac{1}{n}(H_n - H_i).$$

For the second part of the lemma, we use the fact that given the N_i's, the last node ends up in the ith subtree with probability $N_i/(n-1)$. Thus,

$$p_{n,l} = \sum_{j=1}^{4} \sum_{i=0}^{n-1} \frac{i}{n-1} \mathbf{P}(N_j = i) p_{i,l-1},$$

from which we deduce our result by symmetry. □

The basic recurrence of Lemma R1 can now be used to obtain recurrences for the generating function and the moments of D_n. We define

$$\phi_n(t) = \mathbf{E}(e^{tD_n})$$

and

$$\mu_{n,m} = \mathbf{E}(D_n^m) = \phi_n^{(m)}(0).$$

LEMMA R2.

$$\phi_n(t) = \frac{4e^t}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i)\phi_i(t),$$

and, for $m > 0$,

$$\mu_{n,m} = \frac{4}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i) \sum_{j=0}^{m} \binom{m}{j} \mu_{i,j},$$

where $\mu_{n,0}=1$.

Proof of Lemma R2. From Lemma R1,

$$\begin{aligned}
\phi_n(t) &= \sum_{l=1}^{n-1} p_{n,l} e^{tl} = \sum_{l=1}^{n-1} e^{tl} \frac{4}{n(n-1)} \sum_{i=l}^{n-1} i(H_n - H_i) p_{i,l-1} \\
&= \frac{4}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i) \sum_{l=1}^{i} e^{tl} p_{i,l-1} \\
&= \frac{4e^t}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i)\phi_i(t).
\end{aligned}$$

This proves the first recurrence of the lemma. Let us now take $f_n(t) = \sum_{i=1}^{n-1} i(H_n - H_i)\phi_i(t)$. We have

$$\phi_n(t) = \frac{4\,e^t}{n(n-1)} f_n(t),$$

and thus

$$\phi_n^{(m)}(t) = \frac{4\,e^t}{n(n-1)} \sum_{j=0}^{m} \binom{m}{j} f_n^{(j)}(t).$$

Thus,

$$\phi_n^{(m)}(0) = \frac{4}{n(n-1)} \sum_{j=0}^{m} \binom{m}{j} f_n^{(j)}(0) = \frac{4}{n(n-1)} \sum_{j=0}^{m} \binom{m}{j} \sum_{i=1}^{n-1} i(H_n - H_i)\phi_i^{(j)}(0),$$

which concludes the proof of the lemma. □

From Lemma R2, we can conclude, with a little work, the following lemmas:

LEMMA R3.

$$\mu_{n,m} = \sum_{j=0}^{m-1} \binom{m}{j} \mu_{n,j}(-1)^{m-1-j} + \frac{4}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i)\mu_{i,m}.$$

In particular, $\mu_{n,0} = 1$,

$$\mu_{n,1} = 1 + \frac{4}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i)\mu_{i,1},$$

and

$$\mu_{n,2} = 2\mu_{n,1} - 1 + \frac{4}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i)\mu_{i,2}.$$

The recurrences in Lemma R3 are of the following general form:

$$x_n = a_n + \frac{4}{n(n-1)} \sum_{i=1}^{n-1} i(H_n - H_i)x_i, \qquad n \geq 2, \tag{2}$$

where

$$x_1 = 0, \qquad x_2 = a_2. \tag{3}$$

LEMMA R4. *The general solution of recurrences* (2) *and* (3) *is*

$$x_n = a_n + 4 \sum_{j=3}^{n} \frac{\sum_{i=1}^{j-1} i^2(i-1)a_i}{j^2(j-1)^2(j-2)}, \qquad n \geq 3.$$

Proof of Lemma R4. The proof is omitted. □

Lemmas R3 and R4 can now be combined to obtain the moments of D_n. In particular, we have

THEOREM R1. *For* $n \geq 2$,

$$\mu_{n,1} = H_n - \frac{1}{6} - \frac{2}{3n}$$

and

$$\mu_{n,2} = H_n^2 + H_n^{(2)} + \frac{H_n}{6} - \frac{4H_n}{3n} + \frac{7}{9n} - \frac{77}{36}.$$

Also,

$$\mathrm{Var}\,(D_n) = H_n^{(2)} + \frac{1}{2}H_n + \frac{5}{9n} - \frac{4}{9n^2} - \frac{13}{6}.$$

Proof of Theorem R1. For $\mu_{n,1}$, we note that $a_n = 1$, so that simply

$$\begin{aligned}
\mu_{n,1} &= 1 + 4\sum_{j=3}^{n} \frac{\sum_{i=1}^{j-1} i^2(i-1)}{j^2(j-1)^2(j-2)} \\
&= 1 + \frac{1}{3}\sum_{j=3}^{n} \frac{3j-1}{j(j-1)} \\
&= 1 + \frac{1}{3}\sum_{j=3}^{n} \left(\frac{1}{j} + \frac{2}{j-1}\right) \\
&= 1 + \frac{1}{3}\left(H_n - 1 - \frac{1}{2} + 2(H_{n-1} - 1)\right) = H_n - \frac{1}{6} - \frac{2}{3n}.
\end{aligned}$$

From this, we see that in the computation of $\mu_{n,2}$, when we apply Lemma R4, a_n can be set equal to $2\mu_{n,1} - 1 = 2H_n - (4/3)((n+1)/n)$. From Lemmas R3 and R4, we then conclude that

$$\mu_{n,2} = 2H_n - \frac{4}{3}\frac{n+1}{n} + 4\sum_{j=3}^{n} \frac{b_j}{j^2(j-1)^2(j-2)}$$

where

$$\begin{aligned}
b_j &= \sum_{i=1}^{j-1} i^2(i-1)\left(2H_i - \frac{4}{3}\frac{i+1}{i}\right) \\
&= 2\left(\sum_{i=1}^{j} i^2(i-1)H_i - j^2(j-1)H_j\right) - \frac{4}{3}\sum_{i=1}^{j-1} i(i^2-1) \\
&= \frac{j(j-1)(j-2)}{72}\left(12(3j-1)H_j - (33j+29)\right).
\end{aligned}$$

Thus,

$$\begin{aligned}
\mu_{n,2} &= 2H_n - \frac{4}{3}\frac{n+1}{n} + \frac{1}{18}\sum_{j=3}^{n} \frac{12(3j-1)H_j - (33j+29)}{j(j-1)} \\
&= 2H_n - \frac{4}{3}\frac{n+1}{n} + \frac{2}{3}\sum_{j=3}^{n}\left(\frac{1}{j} + \frac{2}{j-1}\right)H_j - \frac{1}{18}\sum_{j=3}^{n}\left(\frac{62}{j-1} - \frac{29}{j}\right) \\
&= 2H_n - \frac{4}{3}\frac{n+1}{n} + \frac{2}{3}\left(\sum_{j=3}^{n}\frac{H_j}{j} + 2\sum_{j=3}^{n}\frac{H_{j-1} + 1/j}{j-1}\right) - \frac{1}{18}\left(33H_n - \frac{37}{2} - \frac{62}{n}\right) \\
&= \frac{1}{6}H_n - \frac{11}{36} + \frac{19}{9n} + \frac{2}{3}\left(\sum_{j=1}^{n}\frac{H_j}{j} - 1 - \frac{3}{4} + 2\sum_{j=2}^{n-1}\frac{H_j}{j} + 2\sum_{j=3}^{n}\frac{1}{j(j-1)}\right) \\
&= \frac{1}{6}H_n - \frac{53}{36} + \frac{19}{9n} - \frac{2}{3}(2H_n + 2) + 2\sum_{j=1}^{n}\frac{H_j}{j} + \frac{4}{3}\left(\sum_{j=2}^{n-1}\frac{1}{j} - \sum_{j=3}^{n}\frac{1}{j}\right) \\
&= \frac{1}{6}H_n - \frac{101}{36} + \frac{19}{9n} - \frac{4H_n}{3n} + H_n^2 + H_n^{(2)} + \frac{4}{3}\left(H_n - \frac{1}{n} - 1 - H_n + 1 + \frac{1}{2}\right) \\
&= \frac{1}{6}H_n - \frac{77}{36} + \frac{7}{9n} - \frac{4H_n}{3n} + H_n^2 + H_n^{(2)}.
\end{aligned}$$

In this derivation, we needed the following identities, some of which can be found in Knuth (1973, pp. 75-77):

$$\sum_{i=1}^{n} H_i = (n+1)H_n - n,$$

$$\sum_{i=1}^{n} iH_i = \frac{n(n+1)}{2} H_n - \frac{n(n-1)}{4},$$

$$\sum_{i=1}^{n} i^2 H_i = \frac{n(n+1)(2n+1)}{6} H_n - \frac{n(n-1)(4n+1)}{36},$$

$$\sum_{i=1}^{n} i^3 H_i = \frac{n^2(n+1)^2}{4} H_n - \frac{n(n^2-1)(3n-2)}{48},$$

$$\sum_{i=1}^{n} \frac{H_i}{i} = \frac{1}{2}(H_n^2 + H_n^{(2)}).$$

Finally, $\mathrm{Var}(D_n)$ is obtained as $\mu_{n,2} - \mu_{n,1}^2$. □

From Theorem R1, we conclude that $\mathbf{E}D_n = \log n + \gamma - \frac{1}{6} + o(1)$ and $\mathrm{Var}(D_n) = \frac{1}{2}\log n + \gamma/2 + \pi^2/6 - \frac{13}{6} + o(1)$, where γ is Euler's constant. Chebyshev's inequality now implies that $D_n/\log n \to 1$ in probability as $n \to \infty$. The resulting upper bound for $\mathbf{P}(D_n/\log n \notin (1-\varepsilon, 1+\varepsilon))$ drops off as $(1+o(1))(2\varepsilon^2 \log n)^{-1}$. The exponential bounding method for the previous section yields better tail bonds, however.

REFERENCES

J. L. BENTLEY (1975), *Multidimensional binary search trees used for associative searching*, Comm. ACM, 18, pp. 509-517.

J. L. BENTLEY, D. F. STANAT, AND E. H. WILLIAMS (1977), *The complexity of fixed-radius near neighbor searching*, Inform. Process. Lett., 6, pp. 209-212.

G. G. BROWN AND B. O. SHUBERT (1984), *On random binary trees*, Math. Oper. Res., 9, pp. 43-65.

H. CHERNOFF (1952), *A measure of asymptotic efficiency for tests of a hypothesis based on the sum of observations*, Annals of Mathematical Statistics, 23, pp. 493-507.

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

——— (1987), *Branching processes in the analysis of the heights of trees*, Acta Inform., 24, pp. 277-298.

——— (1988), *Applications of the theory of records in the study of random trees*, Acta Inform., 26, pp. 123-130.

R. A. FINKEL AND J. L. BENTLEY (1974), *Quad trees: A data structure for retrieval on composite keys*, Acta Inform., 4, pp. 1-9.

P. FLAJOLET, G. GONNET, C. PUECH, AND M. ROBSON (1988), *Analytic variations on quadtrees*, Tech. Report, INRIA, Rocquencourt, France.

G. H. GONNET (1984), *A Handbook of Algorithms and Data Structures*, Addison-Wesley, Reading, MA.

G. M. HUNTER AND K. STEIGLITZ (1979), *Operations on images using quad trees*, IEEE Trans. Pattern Analysis and Machine Intelligence, PAMI-1, pp. 145-153.

D. E. KNUTH (1973), *The Art of Computer Programming, Vol.* 1: *Fundamental Algorithms*, 2nd ed., Addison-Wesley, Reading, MA.

——— (1973), *The Art of Computer Programming, Vol.* 3: *Sorting and Searching*, Addison-Wesley, Reading, MA.

D. T. LEE AND C. K. WONG (1981), *Quintary trees: A file structure for multidimensional data-base systems*, ACM Trans. Database Systems, 5, pp. 339-353.

G. LOUCHARD (1987), *Exact and asymptotic distributions in digital and binary search trees*, Theoretical Informatics and Applications, 21, pp. 479-496.

W. C. LYNCH (1965), *More combinatorial problems on certain trees*, Computer J., 7, pp. 299-302.

H. MAHMOUD AND B. PITTEL (1984), *On the most probable shape of a search tree grown from a random permutation*, SIAM J. Algebraic Discrete Methods, 5, pp. 69-81.

T. H. MERRETT AND E. OTOO (1981), *Multidimensional paging for associative searching*, Tech. Report SOCS 81-18, School of Computer Science, McGill University, Montréal.

J. A. ORENSTEIN (1982), *Multidimensional tries used for associative searching*, Tech. Report School of Computer Science, McGill University, Montréal.

B. PITTEL (1984), *On growing random binary trees*, J. Math. Anal. Appl., 103, pp. 461–480.

C. PUECH AND H. YAHIA (1985), *Quadtress, octrees, hyperoctrees: A unified analytical approach to tree data structures used in graphics, geometric modeling and image processing*, in Proc. Symposium on Computational Geometry, Association for Computing Machinery, New York, pp. 272–280.

R. PYKE (1965), *Spacings*, J. R. Statist. Soc. Ser. B, 7, pp. 395–445.

——— (1975), *Spacings revisited*, Proc. Sixth Berkeley Symposium, 1, pp. 417–427.

R. L. RIVEST (1974), *Analysis of associative retrieval algorithms*, Tech. Report STAN-CS-74-415, Computer Science Department, Stanford University, Stanford, CA.

J. T. ROBINSON (1981), *The* K-B-D *tree: A search structure for large multidimensional dynamic indexes*, Proc. ACM SIGMOD, pp. 10–18.

J. M. ROBSON (1979), *The height of binary search trees*, Austral. Comput. J., 11, pp. 151–153.

H. SAMET (1980), *Deletion in two-dimensional quad trees*, Comm. ACM, 23, pp. 703–710.

——— (1984), *The quadtree and related hierarchical data structures*, Comput. Surveys, 16, pp. 187–260.

R. SEDGEWICK (1985), *Mathematical analysis of combinatorial algorithms*, in Probability Theory and Computer Science, G. Louchard and G. Latouche, eds., Academic Press, London, pp. 123–205.

M. TAMMINEN (1981), *Order preserving extendible hashing and bucket tries*, BIT, 21, pp. 419–435.

——— (1981), *The EXCELL method for efficient geometric access to data*, Acta Polytech. Scand. Math. Comput. Sci. Ser. 34, Helsinki, Finland.

——— (1982), *The extendible cell method for closest point problems*, BIT, 22, pp. 27–41.

J. R. WOODWARK (1982), *The explicit quad tree as a structure for computer graphics*, Comput. J., 25, pp. 235–237.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 833–846, October 1990

005

BOUNDED QUERY CLASSES*

KLAUS W. WAGNER†

Abstract. Polynomial time machines having restricted access to an NP oracle are investigated. Restricted access means that the number of queries to the oracle is restricted or the way in which the queries are made is restricted (e.g., queries made during truth-table reductions). Very different kinds of such restrictions result in the same or comparable complexity classes. In particular, the class $P^{NP}[O(\log n)]$ can be characterized in very different ways. Furthermore, the Boolean hierarchy is generalized in such a way that it is possible to characterize P^{NP} and $P^{NP}[O(\log n)]$ in terms of the generalization.

Key words. polynomial time oracle machines, number of oracle queries, truth-table reducibility, Boolean hierarchy

AMS(MOS) subject classification. 68Q15

1. Introduction. The notion of an oracle machine is a basic tool in complexity theory. Besides other purposes, complexity bounded machines using an oracle have been used to define natural classes which characterize (in term of completeness) the complexity of some interesting problems. In order to characterize the complexity of problems in the area between NP and PSPACE Karp [Kar72] and Meyer and Stockmeyer [MeSt72] (for more details see [Sto77]) introduced the polynomial-time hierarchy using polynomial time machines having access to an NP oracle or to an oracle already defined in such a way, i.e.: $\Delta_0^p = \Sigma_0^p = \Pi_0^p = P$, $\Delta_{k+1}^p = P^{\Sigma_k^p}$, $\Sigma_{k+1}^p = \mathrm{NP}^{\Sigma_k^p}$, $\Pi_{k+1}^p = \mathrm{co}-\Sigma_{k+1}^p$ for $k \geqq 0$, and $\mathrm{PH} = \bigcup_{k \geqq 0} \Sigma_k^p$. In the sequel a lot of problems were shown to be complete for the levels of the polynomial-time hierarchy.

Later it turned out that the levels of this hierarchy do not suffice to characterize the complexity of some interesting problems in PH in terms of completeness. For example, the problem ODD COLORABILITY, i.e., the problem of whether the chromatic number of a given graph is odd is in Δ_2^p and it is NP-hard but it could not be proved that it is complete for Δ_2^p or that it is in NP. This is not very surprising because it is co-NP-hard too and it can easily be solved by a Δ_2^p machine making only $O(\log n)$ (rather than polynomially many) queries to an NP oracle. The class $P^{NP}[O(\log n)]$ of problems which can be solved in this manner was studied first by Papadimitriou and Zachos [PaZa82] and Krentel [Kre86]. In [Wag86] it was proved that ODD COLORABILITY and many other natural problems are complete for the class $P^{NP}[O(\log n)]$ (which was defined there in another way).

In the present paper we investigate different ways to restrict the number of queries to an NP oracle or to restrict the way in which these queries are made. Our main message is: All these different approaches result in the same or strongly related complexity classes. In particular, the class $P^{NP}[O(\log n)]$ is characterized by very different kinds of restricted access to the oracle. Moreover, $P^{NP}[O(\log n)]$ coincides with the class L^{NP} of languages logspace Turing reducible to an NP set. All these results remain valid if we replace NP with any level Σ_k^p of the polynomial-time hierarchy ($k \geqq 2$). This motivates us to define the classes Θ_k^p by $\Theta_0^p = P$ and $\Theta_{k+1}^p = L^{\Sigma_k^p} = P^{\Sigma_k^p}[O(\log n)]$ (remember: $\Delta_{k+1}^p = P^{\Sigma_k^p}$ and $\Sigma_{k+1}^p = \mathrm{NP}^{\Sigma_k^p}$). We suggest to consider the classes Θ_k^p to be constitutional parts of the polynomial-time hierarchy, and we extend

* Received by the editors June 10, 1987; accepted for publication (in revised form) January 3, 1990. A preliminary version of this paper appeared in the proceedings of the 15th ICALP Conference, Tampere, 1988; Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, 1988, pp. 682–696.

† Institut für Informatik, Universität Würzburg, D-8700 Würzburg, Federal Republic of Germany.

the widely accepted conjecture that the polynomial-time hierarchy is proper to the classes Θ_k^p. In particular, we conjecture $\Theta_k^p \neq \Delta_k^p$ for all $k \geqq 2$.

In § 2 we make available the notions of oracle computation with adaptive and parallel queries. The queries of a computation are said to be made in *parallel* if a list of them is formed before any of them is made. Otherwise the queries are called *adaptive*. In § 3 we prove that, for logarithmically bounded constructible r, $r(n)$ adaptive queries to an NP oracle have exactly the same power as $2^{r(n)}-1$ parallel queries. In § 4 we study restrictions to the number of queries not only for one computation but for the whole computation tree of an NP machine and for the whole oracle tree of an NP, P, or L computation. In § 5 we see that making $r(n)$ parallel queries to an NP oracle is closely related to making a truth-table reduction using $r(n)$ instances of an NP set and that different kinds of polynomial time truth-table reducibilities coincide. In § 6 we extend the Boolean hierarchy (see [Köb85], [WeWa85], [CaHe86], [KöScWa87], [CGHHSWW88]) in an obvious way, and we prove that the classes of the extended Boolean hierarchy are closely related to the classes defined by parallel queries and that P^{NP} and $P^{NP}[O(\log n)]$ can be characterized in this way. In § 7 we present an improved criterion for completeness in Θ_2^p.

Finally we note that most of the results in §§ 5 and 6 have been proved independently by Buss and Hay [BuHa88].

2. Oracle computations. For a class C of languages let P^C (NP^C, L^C, respectively) be the class of languages which can be accepted by deterministic polynomial time (nondeterministic polynomial time, deterministic logspace, respectively) bounded Turing machines using an oracle from C. The classes Θ_k^p, Δ_k^p, Σ_k^p, and Π_k^p defined by

$$\Theta_0^p = \Delta_0^p = \Sigma_0^p = \Pi_0^p = P,$$

$$\Theta_{k+1}^p = L^{\Sigma_k^p},\ \Delta_{k+1}^p = P^{\Sigma_k^p},\ \Sigma_{k+1}^p = NP^{\Sigma_k^p}, \qquad \Pi_{k+1}^p = co\text{-}\Sigma_{k+1}^p \quad \text{for } k \geqq 0,$$

build the *polynomial-time hierarchy*. Furthermore, $PH = \bigcup_{k \geqq 0} \Sigma_k^p$. It is not hard to see that $\Theta_1^p = \Delta_1^p = P$, $\Sigma_1^p = NP$, $\Sigma_k^p \cup \Pi_k^p \subseteq \Theta_{k+1}^p \subseteq \Delta_{k+1}^p \subseteq \Sigma_{k+1}^p \cap \Pi_{k+1}^p$ for all $k \geqq 0$ and $PH \subseteq$ PSPACE. It is not known whether this inclusion is proper or whether the polynomial-time hierarchy is proper (even $NP \neq$ PSPACE is not known). It is conjectured that the above inclusions between the classes Δ_k^p, Σ_k^p, and Π_k^p are proper, and we extend this conjecture to include also the classes Θ_k^p.

In the definition of P^C we did not make any restriction to the number of queries allowed or to how the queries to the oracle are made. Now we will consider such restrictions.

Let M be a nondeterministic Turing machine using oracle B. Following Book, Long, and Selman [BoLoSe84] let $Q(M, B, x, p)$ denote the set of all strings queried during a computation p of M on input x when using oracle B. Following Krentel [Kre86] we define for a bounding function $r: \mathbb{N} \to \mathbb{N}$ and a class C of languages:

$NP^C[r]$ is the class of all languages accepted by some nondeterministic polynomial time Turing machine M using an oracle $B \in C$ such that, for all inputs x and all computations p of M on x, $\# Q(M, B, x, p) \leqq r(|x|)$.

Using deterministic polynomial time (logspace, respectively) Turing machines, we define analogously $P^C[r]$ ($L^C[r]$, respectively). Note that for logspace oracle Turing machines the query tape is not subject to the space bound. Obviously, $L^C[r] \subseteq P^C[r] \subseteq NP^C[r]$ for all bounding functions r. Instead of single bounding functions we also use classes of bounding functions. In particular, we use the class Pol of all polynomials and, for a bounding function r, the class $O(r)$. The class $P^{NP}[r]$ is denoted by $P^{NP[r]}$ in [PaZa82], by $Q(r, \text{SAT})$ and $Q(r, \text{NPC})$ in [Gas86] and [AmGa87], and by $Q(r, \text{SAT})$

in [Bei88]. It is evident that $L^{NP}[Pol] = L^{NP}$, $P^{NP}[Pol] = P^{NP}$, and $NP^{NP}[Pol] = NP^{NP}$.

Note that in a $NP^C[r]$ computation every query can depend on the oracle answers to previous queries. We say that such queries are *adaptive* or *made in series.* The oracle queries of a machine are said to be *nonadaptive* or *made in parallel* if a list of all queries is formed before any of them is made. For a class C of languages and a bounding function $r: \mathbb{N} \mapsto \mathbb{N}$ we define:

$NP_{\|}^C$ is the class of all languages accepted by some nondeterministic polynomial time Turing machine M using an oracle $B \in C$ such that, for all inputs x and all computations p of M on x, a list of all queries from $Q(M, B, x, p)$ is formed before any of them is made.

$NP_{\|}^C[r]$ is the class of all languages accepted by some nondeterministic polynomial time Turing machine M using an oracle $B \in C$ such that, for all inputs x and all computations p of M on x, $\# Q(M, B, x, p) \leqq r(|x|)$ and a list of all queries from $Q(M, B, x, p)$ is formed before any of them is made.

Using deterministic polynomial time (logspace) Turing machines we define analogously $P_{\|}^C$ and $P_{\|}^C[r]$ ($L_{\|}^C$ and $L_{\|}^C[r]$, respectively). The case of parallel queries during a logspace computation needs some explanation. We assume that a logspace Turing machine writes the list of queries to the oracle on the query tape and the oracle answers by a list of 0-1 answers to these queries. The machine can read these answers in a one-way mode.

It is clear that the nonadaptive query classes are included in the adaptive query classes with the same bounding function. Furthermore, $L_{\|}^C \subseteq P_{\|}^C \subseteq NP_{\|}^C$ and $L_{\|}^C[r] \subseteq P_{\|}^C[r] \subseteq NP_{\|}^C[r]$ for all bounding functions r. The class $P_{\|}^{NP}[r]$ is denoted by $Q_{\|}(r, SAT)$ in [Bei88]. It is evident that $L_{\|}^{NP}[Pol] = L_{\|}^{NP}$, $P_{\|}^{NP}[Pol] = P_{\|}^{NP}$, and $NP_{\|}^{NP}[Pol] = NP_{\|}^{NP}$.

The characterization of NP^{NP} by polynomially bounded quantifiers in [Sto77] shows that for a nondeterministic polynomial time Turing machine all queries to an NP oracle can be replaced by only one query to another NP oracle. Hence we have Theorem 2.1.

THEOREM 2.1. $NP_{\|}^{NP}[1] = NP_{\|}^{NP} = NP^{NP}[1] = NP^{NP}$.

Figure 1 shows the inclusional relationships between the most interesting classes between $\Sigma_1^p = NP$ and Δ_2^p. It looks rather involved, but in the following sections it will turn out that all classes belonging to the same area (marked off by ▬) in fact coincide. That simplifies our view on the important classes between Σ_1^p and Δ_2^p considerably.

3. Adaptive versus nonadaptive queries. Now it is an interesting problem to discover the exact relationships between the adaptive query classes and nonadaptive query classes, i.e., to find out how many adaptive queries to an NP oracle are sufficient to replace a given number of nonadaptive queries to an NP oracle and vice versa. Because of Theorem 2.1 we must consider only the case of deterministic polynomial time and logspace computations. Thus the problem can be formulated as: how to strengthen the obvious inclusions $P_{\|}^{NP}[r] \subseteq P^{NP}[r]$ and $L_{\|}^{NP}[r] \subseteq L^{NP}[r]$ and how to relate the polynomial time classes with the logspace classes.

An answer for a constant number of queries was given in [Bei88]. It will appear as a special case of a more general result given below. This result says that, for logarithmically bounded constructible functions r, $2^{r(n)} - 1$ parallel queries are exactly as powerful as $r(n)$ adaptive queries. For proving this we start with a simple observation which states that every set $A \in P_{\|}^{NP}[r]$ ($L_{\|}^{NP}[r]$, respectively) is polynomial time (logspace, respectively) truth-table reducible (see § 5) to an NP set using at most $r(n)$

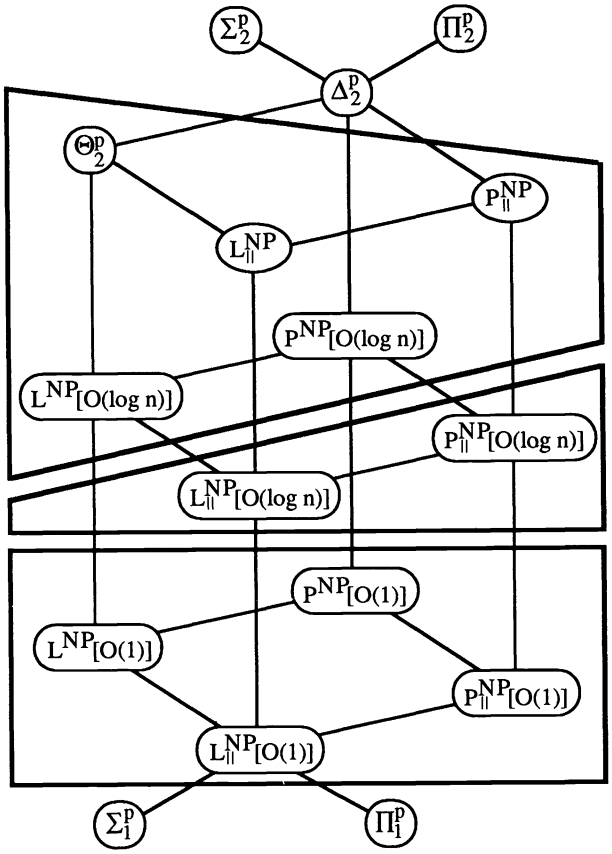


FIG. 1

instances of this NP set for an input of length n. Let c_A be the characteristic function of the set A.

LEMMA 3.1. *Let r be polynomially bounded such that $r(|x|)$ is polynomial time computable. For every $A \in P_{\|}^{NP}[r]$ ($L_{\|}^{NP}[r]$, respectively) there exist a $B \in NP$ and a polynomial time (logspace, respectively) computable function h such that $c_A(x) = h_x(c_B(x_1), \cdots, c_B(x_{r(|x|)}))$ for all x where $h(x) = (N_x, x_1, \cdots, x_{r(|x|)})$ and N_x is a natural encoding of a Boolean circuit computing the Boolean function h_x.*

Proof. For an input x to a $P_{\|}^{NP}[r]$ machine M let $x_1, \cdots, x_{r(|x|)}$ be the queries made by M in parallel. Furthermore, let N_x be the description of a Boolean circuit describing the computation of M on x in an obvious way where N_x has $r(|x|)$ input nodes corresponding to the answers of the oracle to the queries of M. If M is a $L_{\|}^{NP}[r]$ machine, then $(N_x, x_1, \cdots, x_{r(|x|)})$ can be produced by a logspace machine. □

The most important auxiliary result for the main result of this section and for other results of this paper is the following theorem which might be of independent interest. The idea to its proof rests on the *mind change technique* used by Beigel [Bei88] to prove Theorem 3.4 for constant r. The mind change technique can be considered as a refinement of the *compute census and make one more query technique* used by Hemachandra [Hem87] and Kadin [Kad87] (see § 4). For $(a_1, \cdots, a_k), (b_1, \cdots, b_k) \in \{0, 1\}^k$ we write $(a_1, \cdots, a_k) \leqq (b_1, \cdots, b_k)$ if and only if $a_i \leqq b_i$ for $i = 1, \cdots, k$. The k-ary Boolean function h *changes its mind from a to b* if and only if $a \leqq b$ and

$h(a) \neq h(b)$. Let $\mathrm{mc}(h)$ denote the maximum number of mind changes of h, i.e.,

$$\mathrm{mc}(h) = \max\{m: \text{there exists } a_0, a_1, \cdots, a_m \in \{0,1\}^k \text{ such that } a_{i-1} \leqq a_i \text{ and } h(a_{i-1}) \neq h(a_i) \text{ for } i = 1, \cdots, m\}.$$

Obviously, $\mathrm{mc}(h) \leqq k$ for every k-ary Boolean function h.

THEOREM 3.2. (1) *Let $r: \mathbb{N} \mapsto \mathbb{N}$ be polynomially bounded such that $r(|x|)$ is polynomial time computable. For every set A,*

$A \in P_{\|}^{\mathrm{NP}}[r] \Leftrightarrow$ *there exists a set $B \in \mathrm{NP}$ such that $c_B(x, i+1) \leqq c_B(x, i)$ for all x and i and a 0-1-valued polynomial time computable function f such that $c_A(x) \equiv (\max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B\} + f(x)) \bmod 2$.*

(2) *Let $r: \mathbb{N} \mapsto \mathbb{N}$ be polynomially bounded such that $r(|x|)$ is logspace computable. For every set A,*

$A \in L_{\|}^{\mathrm{NP}}[r] \Leftrightarrow$ *there exists a set $B \in \mathrm{NP}$ such that $c_B(x, i+1) \leqq c_B(x, i)$ for all x and i and a 0-1-valued logspace computable function f such that $c_A(x) \equiv (\max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B\} + f(x)) \bmod 2$.*

Proof. (1) For $A \in P_{\|}^{\mathrm{NP}}[r]$ there exists (by Lemma 3.1) an NP-set D and a 0-1-valued polynomial time computable function h such that $c_A(x) = h_x(c_D(x_1), \cdots, c_D(x_{r(|x|)}))$ for all x where $h(x) = (N_x, x_1, \cdots, x_{r(|x|)})$ and N_x is a natural encoding of a Boolean circuit computing the Boolean function h_x. Define $\mathrm{mc}_D(x)$ as the maximum number of mind changes of h_x between $(0, \cdots, 0)$ and $(c_D(x_1), \cdots, c_D(x_{r(|x|)}))$. Obviously, $\mathrm{mc}_D(x) \leqq \mathrm{mc}(h_x) \leqq r(|x|)$ and $c_A(x) \equiv (\mathrm{mc}_D(x) + h_x(0, \cdots, 0)) \bmod 2$. This way of computing $c_A(x)$ is just the basic idea of the mind change technique. Defining

$$B = \{(x, i): \text{there are at least } i \text{ mind changes of } h_x \text{ between } (0, \cdots, 0) \text{ and } (c_D(x_1), \cdots, c_D(x_{r(|x|)}))\}$$

we obtain $\mathrm{mc}_D(x) = \max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B\}$ and $c_B(x, i+1) \leqq c_B(x, i)$ for all i. Defining $f(x) = h_x(0, \cdots, 0)$ it remains to prove that B is in NP. An NP machine M to accept B can be constructed as follows. Given input (x, i), M computes $h(x) = (N_x, x_1, \cdots, x_{r(|x|)})$. If $i > r(|x|)$, then M rejects. Otherwise M guesses nondeterministically $a_0, \cdots, a_i \in \{0,1\}^{r(|x|)}$ and checks whether $a_0 \leqq a_1 \leqq \cdots \leqq a_i$ and $h_x(a_0) \neq h_x(a_1) \neq \cdots \neq h_x(a_i)$. If this fails then M rejects. Otherwise, let $a_i = (b_1, \cdots, b_{r(|x|)})$. Now M checks by consecutive NP computations whether $x_j \in B$ for all $j = 1, \cdots, r(|x|)$ such that $b_j = 1$. If all these computations are accepting then M accepts, otherwise M rejects.

The converse implication is obvious. The logspace case is treated in the same way where we compute $h_x(0, \cdots, 0)$ by running an $L_{\|}^{\mathrm{NP}}[r]$ machine which accepts A on input x with oracle answers $0, \cdots, 0$ (i.e., without using an oracle). □

COROLLARY 3.3. *Let $r: \mathbb{N} \mapsto \mathbb{N}$ be polynomially bounded such that $r(|x|)$ is polynomial time computable. For every set $A \in P_{\|}^{\mathrm{NP}}[r]$ there exists a set $B \in \mathrm{NP}$ such that $c_B(x, i+1) \leqq c_B(x, i)$ for all x and i and*

$$c_A(x) \equiv \max\{i: 1 \leqq i \leqq r(|x|) + 1 \text{ and } (x, i) \in B\} \bmod 2.$$

Proof. For $A \in P_{\|}^{\mathrm{NP}}[r]$ we obtain by Theorem 3.2(1) an NP-set D such that $c_D(x, i+1) \leqq c_D(x, i)$ for all x and i and a 0-1-valued polynomial time computable function f such that

$$c_A(x) \equiv (\max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B\} + f(x)) \bmod 2.$$

The NP-set

$$B = \{(x, 1): f(x) = 1\} \cup \{(x, i + f(x)): 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in D\}$$

has the desired property. □

Now we can establish the main result of this section which has been proved for constant r already in [Bei88].

THEOREM 3.4. *Let $r \in O(\log n)$.*

(1) $P^{NP}[r(n)] = P_{\|}^{NP}[2^{r(n)} - 1]$ *for polynomial time computable r.*

(2) $L^{NP}[r(n)] = L_{\|}^{NP}[2^{r(n)} - 1]$ *for logspace computable r.*

Proof. If we consider the computations of a $P^{NP}[r(n)]$ machine for all possible oracles up to the $r(n)$th query we obtain at most $2^{r(n)} - 1$ different queries which can be computed in polynomial time (remember: $r \in O(\log n)$). Hence $P^{NP}[r(n)] \subseteq P_{\|}^{NP}[2^{r(n)} - 1]$. For the converse inclusion we conclude from Theorem 3.2(1) that for every $A \in P_{\|}^{NP}[2^{r(n)} - 1]$ there exist a set $D \in NP$ and a 0-1-valued function $h \in P$ such that $c_D(x, i+1) \leqq c_D(x, i)$ for all i and

$$c_A(x) \equiv (\max\{i: 1 \leqq i \leqq 2^{r(|x|)} - 1 \text{ and } (x, i) \in D\} + h(x)) \bmod 2.$$

Because of $c_D(x, i+1) \leqq c_D(x, i)$ we can find out this maximum by a binary search with $r(|x|)$ queries to D.

The logspace case is treated in the same manner because the binary search can be carried out by a logspace machine. □

Using Corollary 3.3 we can even convert a polynomial time computation with an NP oracle into a logspace computation with an NP oracle at the price of one more query to the oracle.

THEOREM 3.5. *Let $r(|x|)$ be logspace computable.*

(1) $P_{\|}^{NP}[r(n)] \subseteq L_{\|}^{NP}[r(n)+1]$ *for polynomially bounded r.*

(2) $P^{NP}[r(n)] \subseteq L^{NP}[r(n)+1]$ *for $r \in O(\log n)$.*

COROLLARY 3.6. *Let $r(|x|)$ be logspace computable.*

(1) $P_{\|}^{NP}[r(n) + O(1)] = L_{\|}^{NP}[r(n) + O(1)]$ *for polynomially bounded r.*

(2) $P^{NP}[r(n) + O(1)] = L^{NP}[r(n) + O(1)]$ *for $r \in O(\log n)$.*

Note that it is an open problem whether Corollary 3.6(2) remains true for $r \notin O(\log n)$. We conjecture that it is not true in general since otherwise we would have $P^{NP}[\text{Pol}] = L^{NP}[\text{Pol}]$ which contradicts our conjecture $\Theta_2^p \neq \Delta_2^p$ (see § 2).

For our favorite bounding functions we obtain from Theorem 3.4 and the preceding corollary the following result.

COROLLARY 3.7.

(1) $P^{NP}[O(\log n)] = L^{NP}[O(\log n)] = P_{\|}^{NP} = L_{\|}^{NP}$.

(2) $P_{\|}^{NP}[O(\log n)] = L_{\|}^{NP}[O(\log n)]$.

(3) $P^{NP}[O(1)] = L^{NP}[O(1)] = P_{\|}^{NP}[O(1)] = L_{\|}^{NP}[O(1)]$.

Finally note that the class $P^{NP}[O(1)]$ coincides with the union BH of the classes of the *Boolean hierarchy* defined independently by Köbler, [Köb85], Wechsung, and Wagner [WeWa85] and Cai and Hemachandra [CaHe86]. For the coincidence of $P_{\|}^{NP}[O(1)]$, $P^{NP}[O(1)]$ and BH and other interesting properties of the Boolean hierarchy see [KöScWa87], [Bei88], and [CGHHSWW88].

4. Restricting the number of queries in the computation tree and in the oracle tree. In the preceding sections we have studied restrictions to the number of queries in every single computation. For nondeterministic machines there may be several computations for a given input which build the *computation tree* for this input. Book, Long, and Selman [BoLoSe84] were the first to investigate restrictions on the number of queries in the whole computation tree in order to establish *positive relativizations* of some open problems in complexity theory.

The set of all queries in the computation tree of a nondeterministic Turing machine M using oracle B on input x is denoted by

$$Q(M, B, x) = \bigcup_{p \text{ computation}} Q(M, B, x, p).$$

For a class C of languages and a bounding function r we define:

$\mathrm{NP}^C_{\mathrm{ctree}}[r]$ is the class of all languages which can be accepted by a nondeterministic polynomial time Turing machine M using an oracle $B \in C$ such that $\#Q(M, B, x) \leqq r(|x|)$ for all inputs x.

Note that $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[\mathrm{Pol}]$ was denoted by NP.ALL.DEP(SAT) in [BoLoSe84] and by $\mathrm{NP}_B(\mathrm{NP})$ in [Lon85].

It is obvious that the computation tree of a nondeterministic polynomial time oracle Turing machine cannot include more than $2^{p(n)}$ queries to the oracle (for some polynomial p). Hence $\mathrm{NP}^C_{\mathrm{ctree}}[2^{\mathrm{pol}}] = \mathrm{NP}^C$ for all oracles C. In particular, we have Proposition 4.1.

PROPOSITION 4.1. $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[2^{\mathrm{Pol}}] = \mathrm{NP}^{\mathrm{NP}}$.

Can the class $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[r]$ be characterized also for subexponential bounding functions? The only answer we know was given by Book, Long, and Selman for $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[\mathrm{Pol}]$.

THEOREM 4.2. $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[\mathrm{Pol}] = P^{\mathrm{NP}}$.

Proof. In Corollary 5.4B of [BoLoSe84] it is proved that $\mathrm{NP}^B_{\mathrm{ctree}}[\mathrm{Pol}] = P^B$ if and only if B is NP-hard. Since SAT is NP-hard we obtain $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[\mathrm{Pol}] = \mathrm{NP}^{\mathrm{SAT}}_{\mathrm{ctree}}[\mathrm{Pol}] = P^{\mathrm{SAT}} = P^{\mathrm{NP}}$. □

Unfortunately the proof given in [BoLoSe84] does not work to yield any result for subpolynomially bounding functions r. The problem of characterizing, for example, $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{ctree}}[O(\log n)]$ remains unresolved.

In the remainder of this section we will derive some results on another mode of restricting the number of oracle queries. When we restricted the number of queries in $Q(M, B, x)$ the oracle B was fixed. If we consider all possible oracles, then we also obtain for a deterministic machine several computations for the same input which build the *oracle tree* for that input. The set of all queries in the oracle tree of M on input x is denoted by

$$Q(M, x) = \bigcup_{B \text{ oracle}} Q(M, B, x).$$

Restrictions on $Q(M, x)$ were first investigated by Book, Long, and Selman in [BoLoSe84]. For a class C of languages and a bounding function r we define:

$\mathrm{NP}^C_{\mathrm{otree}}[r]$ is the class of all languages which can be accepted by a nondeterministic polynomial time Turing machine M using an oracle $B \in C$ such that $\#Q(M, x) \leqq r(|x|)$ for all inputs x.

Using deterministic polynomial time (logspace, respectively) machines, we define analogously $P^C_{\mathrm{otree}}[r]$ ($L^C_{\mathrm{otree}}[r]$, respectively). Note that in [BoLoSe84] the classes $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}]$ and $P^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}]$ were denoted by NP.ALL(SAT) and P.ALL(SAT), respectively.

Again, it is obvious that the oracle tree of a (non)deterministic polynomial time oracle machine cannot include more than $2^{p(n)}$ queries to the oracle (for some polynomial p). Hence, $\mathrm{NP}^C_{\mathrm{otree}}[2^{\mathrm{Pol}}] = \mathrm{NP}^C$ and $P^C_{\mathrm{otree}}[2^{\mathrm{Pol}}] = P^C$. Since the number of possible oracle queries during an L^C computation cannot exceed the number of configurations with an empty query tape (which is polynomially bounded) we have $L^C_{\mathrm{otree}}[\mathrm{Pol}] = L^C$. In particular, we have Proposition 4.3.

PROPOSITION 4.3. (1) $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[2^{\mathrm{Pol}}] = \mathrm{NP}^{\mathrm{NP}}$.
(2) $P^{\mathrm{NP}}_{\mathrm{otree}}[2^{\mathrm{Pol}}] = P^{\mathrm{NP}}$.
(3) $L^{\mathrm{NP}}_{\mathrm{otree}}[2^{\mathrm{Pol}}] = L^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}] = L^{\mathrm{NP}}$.

It turns out that for polynomially bounded r all the otree classes defined above are closely related to each other and to the class $P^{\mathrm{NP}}_{\|}[r]$. The inclusions $P^{\mathrm{NP}}_{\|}[r] \subseteq P^{\mathrm{NP}}_{\mathrm{otree}}[r]$ and $L^{\mathrm{NP}}_{\|}[r] \subseteq L^{\mathrm{NP}}_{\mathrm{otree}}[r]$ are obvious. On the other hand we even have $P^{\mathrm{NP}}_{\mathrm{otree}}[r] \subseteq P^{\mathrm{NP}}_{\|}[r]$ because the entire oracle tree can be searched in polynomial time. Hence we have Proposition 4.4.

PROPOSITION 4.4. *For all polynomially bounded r,* $P^{\mathrm{NP}}_{\mathrm{otree}}[r] = P^{\mathrm{NP}}_{\|}[r]$.

However, the classes $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[r]$ and $L^{\mathrm{NP}}_{\mathrm{otree}}[r]$ also coincide with $P^{\mathrm{NP}}_{\|}[r]$ if we take bounding functions $O(r)$ rather than r.

THEOREM 4.5. *For all polynomially bounded r such that* $r(|x|)$ *is logspace computable,* $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[O(r)] = P^{\mathrm{NP}}_{\mathrm{otree}}[O(r)] = L^{\mathrm{NP}}_{\mathrm{otree}}[O(r)] = P^{\mathrm{NP}}_{\|}[O(r)]$.

Proof. Because of Corollary 3.6 we have $P^{\mathrm{NP}}_{\|}[O(r)] \subseteq L^{\mathrm{NP}}_{\|}[O(r)]$, and $L^{\mathrm{NP}}_{\|}[O(r)] \subseteq L^{\mathrm{NP}}_{\mathrm{otree}}[O(r)]$ is obvious.

Thus it remains to prove $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[r] \subseteq P^{\mathrm{NP}}_{\|}[O(r)]$. The main idea in this proof is to precompute for an input x to an $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[r]$ machine the number of all relevant elements in the NP oracle. This can be done with $r(|x|)$ parallel queries. Then, knowing this number, the $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[r]$ computation can be converted into an ordinary NP computation which takes one more adaptive query or, equivalently, which doubles the number of parallel queries. This technique was first used by Hemachandra [Hem87] to prove that $P^{\mathrm{NP}}[O(\log n)]$ includes all sets which are polynomial time truth-table reducible to an NP set, and by Kadin [Kad87] to prove $\mathrm{NP}^B \subseteq P^{\mathrm{NP}}[O(\log n)]$ for every sparse NP set B. Let the set A be accepted by an $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[r]$ machine M using an NP oracle B. The set $D = \{(x, z): z \in B \cap Q(M, x)\}$ is obviously in NP. Define $\mathrm{cen}_D(x) = \#\{z: (x, z) \in D\}$, that is, the number of those queries during the work of M on x which belong to B. The value $\mathrm{cen}_D(x)$ can be computed by the parallel queries $(x, 1)$, $(x, 2), \cdots, (x, r(|x|))$ to the set

$$D' = \{(x, k): \text{there exist at least } k \text{ different } z \text{ such that } (x, z) \in D\}$$

which is obviously in NP.

Now given $(x, \mathrm{cen}_D(x))$ an NP machine M' can simulate the work of M on x as follows. First M' guesses nondeterministically the $s \stackrel{\mathrm{def}}{=} \mathrm{cen}_D(x)$ strings $z_1, \cdots, z_s$ such that $(x, z_i) \in D$ for $i = 1, \cdots, s$ and verifies these guesses by consecutive NP computations. If all guesses are correct then M' can simulate M step by step with the only difference that M' checks whether $z \in \{z_1, \cdots, z_s\}$ if M queries whether z is in B. Let E be the set accepted by M'. Hence, $x \in A \Leftrightarrow (x, \mathrm{cen}_D(x)) \in E$ for all x.

Finally, a new $P^{\mathrm{NP}}_{\|}[2r+1]$ machine M'' can accept A as follows: On input x the machine M'' makes the parallel queries $(x, 1)$, $(x, 2), \cdots, (x, r(|x|))$ to D' and $(x, 0)$, $(x, 1), \cdots, (x, r(|x|))$ to E. If s is the greatest natural number in $[0, r(|x|)]$ such that $(x, s) \in D'$ (i.e., $s = \mathrm{cen}_D(x)$), then M'' accepts if and only if $(x, s) \in E$. Hence, $A \in P^{\mathrm{NP}}_{\|}[2r+1]$. □

COROLLARY 4.6. $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}] = P^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}] = L^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}] = \Theta^p_2$.

The equality $\mathrm{NP}^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}] = P^{\mathrm{NP}}_{\mathrm{otree}}[\mathrm{Pol}]$ has already been proved in [BoLoSe84].

5. Truth-table reducibilities. In this section we will see that polynomial time (logspace, respectively) truth-table reducibilities to NP sets are closely related to polynomial time (logspace, respectively) computations making use of an NP oracle by parallel queries.

Polynomial time and logspace truth table reducibility were introduced by Ladner, Lynch, and Selman in [LaLySe75] and [LaLy76], respectively. Informally, $A \leqq^p_{\mathrm{tt}} B$

($A \leqq_{tt}^{log} B$, respectively) if and only if there is a polynomial time (logspace, respectively) Turing machine computing from x queries $x_1, \cdots, x_s$ to B and a description of a Boolean function h (depending on x) such that $c_A(x) = h(c_B(x_1), \cdots, c_B(x_s))$.

This description is informal because we have not specified *how* the Boolean function h should be described. If it is described by Boolean circuits we really write $A \leqq_{tt}^{p} B$ and $A \leqq_{tt}^{log} B$ (following [LaLySe75] and [LaLy76]). If they are described by Boolean formulas (full truth-tables, respectively) we write $A \leqq_{bf}^{p} B$ and $A \leqq_{bf}^{log} B$ ($A \leqq_{ftt}^{p} B$ and $A \leqq_{ftt}^{log} B$, respectively; cf. [Wag86] and [KöScWa87]). If, for a bounding function r, the number s of queries to B is bounded by $r(|x|)$ then we write $A \leqq_{r\text{-}tt}^{p} B$ ($A \leqq_{r\text{-}tt}^{log} B$, $A \leqq_{r\text{-}bf}^{p} B, \cdots$). For every reducibility $\leqq$ defined above and every class C of languages we denote by $\leqq(C)$ the class of all languages A which are $\leqq$-reducible to a set from C. Obviously, $\leqq_{ftt}^{p}(C) \subseteq \leqq_{bf}^{p}(C) \subseteq \leqq_{tt}^{p}(C)$ and $\leqq_{ftt}^{log}(C) \subseteq \leqq_{bf}^{log}(C) \subseteq \leqq_{tt}^{log}(C)$, the same holds true for the r-bounded classes. In fact, for $C = NP$, some of these classes coincide with the corresponding bounded parallel query classes.

THEOREM 5.1. *Let r be polynomially bounded.*

(1) $\leqq_{r\text{-}tt}^{log}(NP) = \leqq_{r\text{-}bf}^{p}(NP) = \leqq_{r\text{-}tt}^{p}(NP) = P_{\|}^{NP}[r]$ *for polynomial time computable r.*

(2) $\leqq_{r\text{-}bf}^{log}(NP) = L_{\|}^{NP}[r]$ *for logspace computable r.*

Proof. (1) The inclusions $\leqq_{r\text{-}tt}^{log}(NP) \cup \leqq_{r\text{-}bf}^{p}(NP) \subseteq \leqq_{r\text{-}tt}^{p}(NP) \subseteq P_{\|}^{NP}[r]$ are obvious; the inclusions $P_{\|}^{NP}[r] \subseteq \leqq_{r\text{-}bf}^{p}(NP)$ and $P_{\|}^{NP}[r] \subseteq \leqq_{r\text{-}tt}^{log}(NP)$ are consequences of Theorem 3.2(1).

(2) The inclusion $\leqq_{r\text{-}bf}^{log}(NP) \subseteq L_{\|}^{NP}[r]$ follows from the fact that Boolean formulas can be evaluated in logspace [Lyn77] (whereas Boolean circuits can probably not because the circuit value problem is P-complete [Lad75]). The inclusion $L_{\|}^{NP}[r] \subseteq \leqq_{r\text{-}bf}^{log}(NP)$ is a consequence of Theorem 3.2(2). □

By this theorem, Corollary 3.7, and the well-known result by Ladner and Lynch [LaLy76] that logspace Turing reducibility is identical with logspace truth-table reducibility, we obtain Corollary 5.2.

COROLLARY 5.2. $\leqq_{bf}^{log}(NP) = \leqq_{tt}^{log}(NP) = \leqq_{bf}^{p}(NP) = \leqq_{tt}^{p}(NP) = L_{\|}^{NP} = P_{\|}^{NP} = L^{NP} = \Theta_2^p$.

The equalities $L^{NP} = L_{\|}^{NP} = L^{NP}[O(\log n)]$ might be surprising. Remember that the number of different oracle queries during a deterministic logspace bounded computation on input x cannot exceed the number of possible configurations with an empty query tape. Thus cycling through all these polynomially many configurations (let p be a suitable polynomial) we can precompute all possible oracle queries.

The inclusion $\leqq_{tt}^{p}(NP) \subseteq P^{NP}[O(\log n)]$ was proved by Hemachandra [Hem87], and the relationship $\leqq_{bf}^{log}(NP) = \leqq_{tt}^{log}(NP) = \leqq_{bf}^{p}(NP) = \leqq_{tt}^{p}(NP) = L^{NP}[O(\log n)] = P^{NP}[O(\log n)]$ was proved independently by Buss and Hay [BuHa88]. The equality $\leqq_{bf}^{p}(NP) = P^{NP}[O(\log n)]$ can already be concluded from [Kre86] and [Wag86] in a very unusual way: Krentel proved that the problem of whether a given graph has a maximum clique size divisible by a given k is complete for $P^{NP}[O(\log n)]$. In [Wag86] the same was proved for the class $\leqq_{bf}^{p}(NP)$. Part of the motivation for this work comes from the dissatisfaction with this proof that $\leqq_{bf}^{p}(NP) = P^{NP}[O(\log n)]$. We feel that the results presented here give more insight into why these classes coincide.

Corollaries 3.7 and 5.2 show that the question of whether $\Delta_2^p = \Theta_2^p$ is identical with the question of whether during a polynomial time computation logarithmically many adaptive queries to an NP-oracle are as powerful as polynomially many adaptive queries. Moreover, it is identical with the question of whether during a polynomial time computation polynomially many parallel queries to an NP-oracle are as powerful as polynomially many adaptive queries. This supports our conjecture $\Delta_2^p \neq \Theta_2^p$.

We conclude this section with a result on the classes $\leqq_{ftt}^{p}(NP)$ and $\leqq_{ftt}^{log}(NP)$. On

the one side, a full truth-table of polynomial size describes a Boolean function with $O(\log n)$ variables. Hence, $\leqq^{\log}_{\mathrm{ftt}}(\mathrm{NP}) \subseteq \leqq^{p}_{\mathrm{ftt}}(\mathrm{NP}) \subseteq P_{\|}^{\mathrm{NP}}[O(\log n)]$. On the other side, from Corollary 3.3 we easily obtain $P_{\|}^{\mathrm{NP}}[O(\log n)] \subseteq \leqq^{\log}_{\mathrm{ftt}}(\mathrm{NP})$. Consequently, we have Theorem 5.3.

THEOREM 5.3. [Köb87] $\leqq^{\log}_{\mathrm{ftt}}(\mathrm{NP}) = \leqq^{p}_{\mathrm{ftt}}(\mathrm{NP}) = P_{\|}^{\mathrm{NP}}[O(\log n)]$.

Thus it does not seem to be very likely that $\leqq^{p}_{\mathrm{ftt}}(\mathrm{NP})$ and $\leqq^{p}_{\mathrm{bf}}(\mathrm{NP})$ coincide because this means $P_{\|}^{\mathrm{NP}}[O(\log n)] = P_{\|}^{\mathrm{NP}}[\mathrm{Pol}]$ which, by a result in [Wag87], implies the collapse of the polynomial-time hierarchy to Θ_3^p.

6. The extended Boolean hierarchy. From the various definitions and equivalent formulations of the classes $\mathrm{NP}(k)$ of the Boolean hierarchy given in [Köb85], [WeWa85], [CaHe86], [KöScWa87], [Bei88], and [CGHHSWW88], we take the following because it can easily be used to extend the Boolean hierarchy. For $k \geqq 1$ we define

$$A \in \mathrm{NP}(k) \Leftrightarrow \text{there exists a set } B \in \mathrm{NP} \text{ such that } c_B(x, i+1) \leqq c_B(x, i) \text{ for all } i$$
$$\text{and } c_A(x) \equiv \max\{i: 1 \leqq i \leqq k \text{ and } (x, i) \in B\} \bmod 2$$

and $BH = \bigcup_{k \geqq 1} \mathrm{NP}(k)$. Obviously, $\mathrm{NP} = \mathrm{NP}(1)$, and it was proved that $P_{\|}^{\mathrm{NP}}[k] = P^{\mathrm{NP}(k)}[1]$ and $\mathrm{NP}(k) \cup \text{co-NP}(k) \subseteq P_{\|}^{\mathrm{NP}}[k] \subseteq \mathrm{NP}(k+1) \cap \text{co-NP}(k+1)$ (cf. [KöScWa87], [Bei88]).

Since we also want to have such interesting relationships for nonconstant bounding functions r we extend the Boolean hierarchy defining

$$A \in \mathrm{NP}(r) \Leftrightarrow \text{there exists a set } B \in \mathrm{NP} \text{ such that } c_B(x, i+1) \leqq c_B(x, i) \text{ for all } i$$
$$\text{and } c_A(x) \equiv \max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B\} \bmod 2.$$

From Theorem 3.2 and Corollary 3.3 we immediately obtain the following two theorems.

THEOREM 6.1. *Let r be polynomially bounded.*
(1) $P_{\|}^{\mathrm{NP}}[r] = P^{\mathrm{NP}(r)}[1]$ *if $r(|x|)$ is polynomial time computable.*
(2) $L_{\|}^{\mathrm{NP}}[r] = L^{\mathrm{NP}(r)}[1]$ *if $r(|x|)$ is logspace computable.*

THEOREM 6.2. *For every polynomially bounded function r such that $r(|x|)$ is logspace computable,*

$$\mathrm{NP}(r) \cup \text{co-NP}(r) \subseteq L_{\|}^{\mathrm{NP}}[r] \subseteq P_{\|}^{\mathrm{NP}}[r] \subseteq \mathrm{NP}(r+1) \cap \text{co-NP}(r+1).$$

COROLLARY 6.3. *For every polynomially bounded function r such that $r(|x|)$ is polynomial time computable,*

$$\mathrm{NP}(r + O(1)) = P_{\|}^{\mathrm{NP}}[r + O(1)].$$

In particular, we have the following result, which has been proved independently by Buss and Hay [BuHa88].

COROLLARY 6.4. $\mathrm{NP}(\mathrm{Pol}) = \Theta_2^p$.

What about superpolynomially bounding functions? Here we know only a single result for 2^{Pol}.

THEOREM 6.5. $\mathrm{NP}(2^{\mathrm{Pol}}) = \Delta_2^p$.

Proof. The inclusion $\mathrm{NP}(2^{\mathrm{Pol}}) \subseteq P^{\mathrm{NP}}$ is proved by a binary search argument. For the converse inclusion it suffices to prove that some P^{NP} complete problem is in $\mathrm{NP}(2^{\mathrm{Pol}})$. We do so for the problem MAX SAT ODD of whether the maximum satisfying assignment to the variables of a Boolean formula is odd (see [Kre86], [Wag86]). Let F be a Boolean formula with n variables and let bin be the obvious correspondence

between the integers of the interval $[0, 2^n - 1]$ and the n-tuples of zeros and ones:

$$F \in \text{MAX SAT ODD} \Leftrightarrow \max\{k: F(\text{bin}(k)) = 1\} \text{ is odd}$$
$$\Leftrightarrow \max\{k: \exists m(m \geqq k \text{ and } F(\text{bin}(m)) = 1)\} \text{ is odd}$$
$$\Leftrightarrow \max\{k: 0 \leqq k \leqq 2^n - 1 \text{ and } (F, k) \in B\} \text{ is odd}$$

where $B = \{(F, k): \exists m(m \geqq k \text{ and } F(\text{bin}(m)) = 1)\}$. Obviously, $B \in \text{NP}$ and

$$c_B(x, k+1) \leqq c_B(x, k). \qquad \square$$

It is an interesting consequence of Corollary 6.4 and Theorem 6.5 that the question of whether $\Theta_2^p = \Delta_2^p$ is identical with the question $\text{NP(Pol)} = \text{NP}(2^{\text{Pol}})$.

Finally let us consider some modifications of $\text{NP}(r)$. For an arbitrary class C, the class $C(r)$ is defined as $\text{NP}(r)$ but replacing NP with C in the definition of $\text{NP}(r)$.

THEOREM 6.6. (1) *For every exponentially bounded polynomial time computable function r, $P(r) = P$.*

(2) *For every polynomially bounded logspace computable function r, $\text{NL}(r) = \text{NL}$.*

(3) *For every polynomially bounded logspace computable function r, $L(r) = L$.*

Proof. (1) For every A from $P(2^{\text{Pol}})$ the property "$x \in A$" can be tested by a binary search with polynomially many queries to a P set. Hence, A is in P.

(2) For every A from NL(Pol) the property "$x \in A$" can be tested by querying (x, i) to an NL set for $1 = 1, \cdots, p(|x|)$ (where p is a suitable polynomial). Hence, A is in L^{NL}. Because of the results of Szelepcsényi and Immerman [Sze87], [Imm88] that NL is closed under complement the logspace oracle hierarchy collapses to NL.

(3) For every A from $L(\text{Pol})$ the property "$x \in A$" can be tested by querying (x, i) to an L set for $1 = 1, \cdots, p(|x|)$ (where p is a suitable polynomial). Hence, A is in L. $\square$

Thus for superpolynomial functions r we only know $\text{NP}(2^{\text{Pol}}) = \Delta_2^p$ and $P(2^{\text{Pol}}) = P$. We do not know characterizations for $\text{NL}(2^{\text{Pol}})$ and $L(2^{\text{Pol}})$. The situation changes a little if we modify the definition of $C(r)$ in such a way that we do not require $c_B(x, k+1) \leqq c_B(x, k)$. Let $C'(r)$ be the class defined in this way.

THEOREM 6.7. (1) *For every exponentially bounded function r with polynomial time computable $r(|x|)$, $\text{NP}'(r) = \text{NP}(r)$.*

(2) *For every polynomially bounded polynomial time computable function r, $P'(r) = P$.*

(3) *For every polynomially bounded logspace computable function r, $\text{NL}'(r) = \text{NL}$.*

(4) *For every polynomially bounded logspace computable function r, $L'(r) = L$.*

(5) $\text{NP}'(2^{\text{Pol}}) = P'(2^{\text{Pol}}) = \text{NL}'(2^{\text{Pol}}) = L'(2^{\text{Pol}}) = \Delta_2^p$.

Proof. (1) Obviously, $\text{NP}(r) \subseteq \text{NP}'(r)$. For an A from $\text{NP}'(r)$ there exists an NP set B such that $c_A(x) \equiv \max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B\} \bmod 2$. Defining the NP set

$$B' = \{(x, i): \text{there exists a } k \text{ such that } i \leqq k \leqq r(|x|) \text{ and } (x, k) \in B\}$$

we obtain $c_A(x) \equiv \max\{i: 1 \leqq i \leqq r(|x|) \text{ and } (x, i) \in B'\} \bmod 2$ and $c_{B'}(x, k+1) \leqq c_{B'}(x, k)$. Statements (3) and (4) can be proved as the corresponding statements in Theorem 6.6. Statement (2) can be proved as statement (4).

(5) Because of $\text{NP}'(2^{\text{Pol}}) = \text{NP}(2^{\text{Pol}}) = \Delta_2^p$ it remains to prove $\Delta_2^p \subseteq L'(2^{\text{Pol}})$. This is done as in the proof of Theorem 6.5 but defining $B = \{(F, k): F(\text{bin}(k)) = 1\}$ which is in L (cf. [Lyn77]). $\square$

7. Complete problems for Q_2^p. It is a simple observation that, for every function r, the class $P_{\|}^{\text{NP}}[r(\text{Pol})]$ is closed under polynomial time many-one reducibility. Thus it is interesting to look for natural complete sets for these classes. The equality $P_{\|}^{\text{NP}}[r(\text{Pol}) + O(1)] = \text{NP}(r(\text{Pol}) + O(1))$ (Corollary 6.3) provides us with a useful tool

to prove completeness results. We restrict ourselves to the case $\Theta_2^p = \mathrm{NP(Pol)}$; for other bounds one can proceed analogously.

THEOREM 7.1. *Let B be logspace many-one complete for* NP *and let A be from Θ_2^p. If there exists a logspace computable function f such that*

$$\max\{i: 1 \leqq i \leqq s \text{ and } x_i \in B\} \text{ is odd} \Leftrightarrow f(x_1, \cdots, x_s) \in A$$

for all $(x_1, \cdots, x_s)$ such that $c_B(x_{i+1}) \leqq c_B(x_i)$ for $i = 1, \cdots, s-1$ then A is logspace many-one complete for Θ_2^p.

Proof. Let C be an arbitrary set from Θ_2^p. Because of Corollary 6.4 there exist a polynomial p and an NP set D such that $c_D(x, k+1) \leqq c_D(x, k)$ for all x and i and $c_C(x) \equiv \max\{i: 1 \leqq i \leqq p(|x|) \text{ and } (x, i) \in D\}$. Furthermore, there exists a logspace computable function g such that $z \in D \Leftrightarrow g(z) \in B$ for all z. Hence $c_B(g(x, i+1)) \leqq c_B(g(x, i))$ for all x and i and

$$x \in C \Leftrightarrow \max\{i: 1 \leqq i \leqq p(|x|) \text{ and } g(x, i) \in B\}$$

$$\Leftrightarrow f(g(x, 1), \cdots, g(x, p(|x|))) \in A.$$

Obviously, $h(x) = f(g(x, 1), \cdots, g(x, p(|x|)))$ is logspace computable. □

In [Wag86] a similar theorem has been proved where "logspace" is replaced with "polynomial time." Several optimization problems of the form "is the optimum of ... odd?" were proved there to be polynomial time many-one complete for Θ_2^p (appearing there as $P_{\mathrm{bf}}^{\mathrm{NP}}$), for example, the problems ODD CLIQUE (see also [Kre86]), ODD MAX 3SAT, ODD COLOUR, ODD INDEPENDENT SET, and ODD VERTEX COVER. A close inspection of the proofs there yields that even the stronger assumptions of Theorem 7.1 are fulfilled. Consequently, all the problems mentioned above are logspace many-one complete for Θ_2^p. For further complete sets for Θ_2^p see [Kre86], [Wag86], [KöScWa87], and [CGHHSWW88].

8. Conclusions. We have seen that very different ways to define restrictions to the use of NP oracles during NP, P, or L computations result in the same or a closely related classes. In particular, Θ_2^p seems to be a very natural class in the interesting area between NP and Δ_2^p because it can be characterized in very different ways. We summarize the most interesting characterizations of Θ_2^p.

THEOREM 8.1.

$$\begin{aligned}\Theta_2^p &= L^{\mathrm{NP}} = L^{\mathrm{NP}}[O(\log n)] = P^{\mathrm{NP}}[O(\log n)] = L_{\|}^{\mathrm{NP}} = P_{\|}^{\mathrm{NP}} = \mathrm{NP(Pol)} \\ &= L_{\mathrm{otree}}^{\mathrm{NP}}[2^{\mathrm{Pol}}] = L_{\mathrm{otree}}^{\mathrm{NP}}[\mathrm{Pol}] = P_{\mathrm{otree}}^{\mathrm{NP}}[\mathrm{Pol}] = \mathrm{NP}_{\mathrm{otree}}^{\mathrm{NP}}[\mathrm{Pol}] \\ &= \leqq_{\mathrm{bf}}^{\log}(\mathrm{NP}) = \leqq_{\mathrm{tt}}^{\log}(\mathrm{NP}) = \leqq_{\mathrm{bf}}^{p}(\mathrm{NP}) = \leqq_{\mathrm{tt}}^{p}(\mathrm{NP}).\end{aligned}$$

Furthermore, Θ_2^p has very natural complete problems. The same (in particular, Theorem 8.1) remains valid for all classes Θ_k^p (where NP is to replace with Σ_{k-1}^p, the proofs are essentially the same). Hence it seems to be natural to consider the classes Θ_k^p to be constitutional parts of the polynomial-time hierarchy. We extend the widely accepted conjecture that all classes of the polynomial-time hierarchy are different also to the classes Θ_k^p. However, while we know [Sto77] that $\Sigma_k^p = \Delta_k^p$, $\Sigma_k^p = \Pi_k^p$, or $\Sigma_k^p = \Delta_{k+1}^p$ for $k \geqq 1$ imply the collapse of the polynomial-time hierarchy to that level we do not know any interesting consequence of $\Theta_k^p = \Delta_k^p$.

We conclude this section by mentioning further work on bounded query classes. A survey on this topic can be found in [Wag88a]. Kadin proved in [Kad88] that $\mathrm{NP}(k) = \text{co-NP}(k)$ for some $k \geqq 1$ implies the collapse of the polynomial time hierarchy to Θ_3^p. This result has been improved and generalized in [Wag87] as follows. Let r be a monotonic function such that $r(n) \leqq n^\alpha$ for some $\alpha < 1$ and that $r(|x|)$ is polynomial

time computable. If $\mathrm{NP}(r) = \text{co-NP}(r)$, then the polynomial time hierarchy collapses to $P_{\|}^{\Sigma_2^p}[O(r(\mathrm{Pol}))]$ (for constant r this is the Boolean closure of Σ_2^p). By Theorem 6.2, if $P_{\|}^{\mathrm{NP}}[r] = P_{\|}^{\mathrm{NP}}[r+1]$ then the polynomial time hierarchy collapses to $P_{\|}^{\Sigma_2^p}[O(r(\mathrm{Pol}))]$. Thus one more query to an NP set gives a polynomial time computation more power unless the polynomial time hierarchy collapses.

Acknowledgments. I would like to thank the anonymous referees for many extremely valuable suggestions. I am grateful to Pekka Orponen, Helsinki, for presenting the results of this paper at 15th ICALP Conference in Tampere.

REFERENCES

[AmGa87] A. AMIR AND W. I. GASARCH, *Polynomial terse sets*, in Proc. 2nd IEEE Conference on Structure in Complexity Theory, 1987, pp. 22–27; Inform. and Computation, 77 (1988), pp. 37–56.

[Bei88] R. J. BEIGEL, *Bounded queries to* SAT *and the Boolean hierarchy*, Theoret. Comput. Sci., (1990), to appear.

[BoLoSe84] R. V. BOOK, T. J. LONG, AND A. L. SELMAN, *Quantitative relativization of complexity classes*, SIAM J. Comput., 13 (1984), pp. 461–487.

[BuHa88] S. R. BUSS AND L. HAY, *On truth-table reducibility to* SAT *and the difference hierarchy over* NP, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 224–233.

[CaHe86] J. CAI AND L. A. HEMACHANDRA, *The Boolean hierarchy: hardware over* NP, in Proc. 1st Conference on Structure in Complexity Theory, 1986; Lecture Notes in Computer Science, Vol. 223, Springer-Verlag, Berlin, New York, 1986, pp. 105–124.

[CGHHSWW88] J. CAI, T. GUNDERMANN, J. HARTMANIS, L. HEMACHANDRA, V. SEWELSON, K. W. WAGNER, AND G. WECHSUNG, *The Boolean hierarchy* I: *Structural properties*, SIAM J. Comput., 17 (1988), pp. 1232–1252.

[Gas86] W. GASARCH, *The complexity of optimization functions*, Tech. Report 1652, Department of Computer Science, University of Maryland, College Park, MD, 1986.

[Hem87] L. A. HEMACHANDRA, *The strong exponential hierarchy collapses*, in Proc. 19th STOC Conference, Association for Computing Machinery, 1987, pp. 110–122.

[Imm88] N. IMMERMAN, *Nondeterministic space is closed under complementation*, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 112–115.

[Kad87] J. KADIN, $P^{\mathrm{NP}}[\log n]$ *and sparse Turing-complete sets for* NP, in Proc. 2nd IEEE Conference on Structure in Complexity Theory, 1987, pp. 33–40.

[Kad88] ———, *The polynomial time hierarchy collapses if the Boolean hierarchy collapses*, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 278–292.

[Kar72] R. M. KARP, *Reducibilities among combinatorial problems*, in Complexity of Computer Computations, Plenum Press, New York, 1972, pp. 85–103.

[Köb85] J. KÖBLER, *Untersuchung verschiedener polynomieller Reduktionsklassen von* NP, thesis, University of Stuttgart, Stuttgart, FRG, 1985.

[Köb87] ———, personal communication.

[KöScWa87] J. KÖBLER, U. SCHÖNING, AND K. W. WAGNER, *The difference and the truth-table hierarchies for* NP, RAIRO Inform. Theor., 21 (1987), pp. 419–435.

[Kre86] M. W. KRENTEL, *The complexity of optimization problems*, in Proc. 18th Ann. STOC, Association for Computing Machinery, 1986, pp. 69–76.

[Lad75] R. E. LADNER, *The circuit value problem is logspace complete for P*, SIGACT News, 7 (1975), pp. 18–20.

[LaLy76] R. E. LADNER AND N. A. LYNCH, *Relativizations of questions about log space computability*, Math. Systems Theory, 10 (1976), pp. 19–32.

[LaLySe75] R. E. LADNER, N. A. LYNCH, AND A. L. SELMAN, *A comparison of polynomial time reducibilities*, Theoret. Comput. Sci., 1 (1975), pp. 103–123.

[Lon85] T. J. LONG, *On restricting the size of oracles compared with restricting access to oracles*, SIAM J. Comput., 14 (1985), pp. 585–597.

[Lyn77] N. A. LYNCH, *Logspace recognition and translation of parenthesis languages*, J. Assoc. Comput. Mach., 24 (1977), pp. 583–590.

[MeSt72] A. R. MEYER AND L. J. STOCKMEYER, *The equivalence problem for regular expressions with squaring requires exponential space*, in Proc. 13th Annual IEEE Symposium on Switching and Automata Theory, 1972, pp. 125–129.

[PaZa82] C. H. PAPADIMITRIOU AND S. K. ZACHOS, *Two remarks on the power of counting*, in Proc. 6th GI Conference on Theoretical Computer Science, Lecture Notes in Computer Science, Vol. 145, Springer-Verlag, Berlin, New York, 1983, pp. 269–275.

[Sto77] L. J. STOCKMEYER, *The polynomial time hierarchy*, Theoret. Comput. Sci., 3 (1977), pp. 1–22.

[Sze87] R. SZELEPCSÉNYI, *The method of forcing for nondeterministic automata*, EATCS Bulletin, 33 (1987), pp. 96–100.

[Wag86] K. W. WAGNER, *More complicated questions about maxima and minima, and some closures of* NP, in Proc. 13th ICALP, Lecture Notes in Computer Science, Vol. 226, Springer-Verlag, Berlin, New York, 1986, pp. 434–443; Theoret. Comput. Sci., 51 (1987), pp. 53–80.

[Wag87] ———, *Number-of-query hierarchies*, Report No. 158, Institute of Mathematics, University of Augsburg, Augsburg, FRG, 1987.

[Wag88a] ———, *Bounded query computations*, in Proc. 3rd IEEE Conference on Structure in Complexity Theory, 1988, pp. 260–277.

[Wag88b] ———, *On restricting the access to an* NP*-oracle*, in Proc. 15th ICALP, Lecture Notes in Computer Science, Vol. 317, Springer-Verlag, Berlin, New York, 1988, pp. 682–696.

[WeWa85] G. WECHSUNG AND K. W. WAGNER, *On the Boolean closure of* NP, manuscript, 1985. (Extended abstract by: G. Wechsung, On the Boolean closure of NP, in Proc. Conference on Fundamentals of Computation Theory, Cottbus, 1985, Lecture Notes in Computer Science, Vol. 199, Springer-Verlag, Berlin, New York, 1985, pp. 485–493.)

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 847-867, October 1990

006

ON THREE-DIMENSIONAL PACKING*

KEQIN LI† AND KAM-HOI CHENG†

Abstract. The three-dimensional packing problem is discussed in this paper. The problem is a generalization of the one- and two-dimensional packing problems. It is demonstrated that some basic packing strategies such as NFDH and FFDH for two-dimensional packing have unbounded worst-case performance ratios in the three-dimensional case. Let $r(A)$ denote the asymptotic performance bound of an approximation algorithm A. An approximation algorithm G is developed, and it is shown that $4.333 \leqq r(G) \leqq 4.571$. The algorithm is improved to algorithm C and it is proven that $r(C) = 3.25$. For the special case when all boxes have square bottoms, the two algorithms are adapted to algorithms G_1 and C_1, respectively, with $r(G_1) = 4$ and $r(C_1) = 2.6875$. For the case when both sides of the bottom of a box are no larger then $1/m$, two families of algorithms, $G_m^*(m \geqq 3)$ and $C_m^*(m \geqq 2)$, are presented. It is shown that $r(G_m^*) = m/(m-2)$ and $r(C_m^*) = (m+1)/(m-1)$. Similarly, when these small bottom boxes have square bottoms, there exist two families of algorithms, $G_m(m \geqq 2)$ and $C_m(m \geqq 2)$, such that $r(G_m) = (m/(m-1))^2$ and $r(C_m) = ((m+1)/m)^2$.

Key words. approximation algorithm, asymptotic performance bound, FFDH, NFDH, NP-hard, three-dimensional packing

AMS(MOS) subject classification. 68Q25

1. Introduction.

The *three-dimensional packing* (3D packing) problem is to pack a set of rectangular boxes $I = \{b_1, b_2, \cdots, b_n\}$ into a rectangular box B with a fixed size bottom and unbounded height such that the height of the packing is minimized. A box b_i is specified as a triplet $b_i = (x_i, y_i, z_i)$ in which x_i, y_i, and z_i are referred to as its *width*, *depth*, and *height*, respectively. We require that all boxes be packed into B orthogonally and oriented in all three dimensions. For a rigorous definition, consult Appendix A. The 3D packing problem is first introduced in [12] as a model of job scheduling in partitionable mesh connected systems (PMCS). The bottom of the box B represents a PMCS of size $w \times l$ and the infinite height stands for the time dimension. Each box $b_i = (x_i, y_i, z_i)$ in I is interpreted as a job which requires a submesh of size $x_i \times y_i$ and z_i units of execution time. The 3D packing problem is then equivalent to scheduling a set of jobs on a PMCS nonpreemptively so that the finish time is minimized. Readers are referred to [13] and [14] for more details about job scheduling in PMCS. Without loss of generality (w.l.o.g.) we may assume that $w = l = 1$ and $x_i, y_i \in (0, 1]$, $1 \leqq i \leqq n$. Therefore an instance of the 3D packing problem only consists of a set of boxes I.

Clearly, the 3D packing problem is a natural generalization of the classical one- and two-dimensional packing problems. The following problems are all special cases of the 3D packing problem and all of them have been extensively studied [6].

- *Bin packing* [10], [17]—if all rectangles in I have width 1 and the same height z.
- *Multiprocessor scheduling* [9], [16]—when all rectangles in I have width 1 and the same depth y.
- *Two-dimensional bin packing* [4]—if all rectangles in I have the same height z.
- *Rectangle packing* [1]-[3], [5]—when all rectangles in I have depth 1.

All these problems are NP-hard, hence the 3D packing problem is also NP-hard for obvious reasons. Thus it is unlikely that we can devise an algorithm which is always

* Received by the editors May 22, 1989; accepted for publication (in revised form) December 28, 1989. This research is based in part on work supported by the Texas Advanced Research Program under grant 1028-ARP.

† Department of Computer Science, University of Houston, Houston, Texas 77204-3475.

able to produce an optimal packing for a given set of boxes in polynomial time. An important practical approach to deal with NP-hard optimization problems is to design and analyze approximation algorithms of polynomial time complexity which generate near-optimal solutions [8]. Suppose A is an approximation algorithm to solve the 3D packing problem. Let $A(I)$ denote the height of the packing produced by A for an instance I and $\mathrm{OPT}(I)$ be the height of an optimal packing of I. A commonly used performance measure in packing problems is the *asymptotic performance bound* which characterizes the behaviour of A when the ratio of $\mathrm{OPT}(I)$ to the maximum box height goes to infinity [5]. If there exist constants α and β such that for all I in which no box has height exceeding Z, $A(I) \leqq \alpha \cdot \mathrm{OPT}(I) + \beta Z$, then α is called *an* asymptotic performance bound of algorithm A. Furthermore, if for any small $\varepsilon > 0$ and any large $N > 0$, there exists an instance I such that $A(I) > (\alpha - \varepsilon) \cdot \mathrm{OPT}(I)$ and $\mathrm{OPT}(I) > N$, then α is tight and denoted as $r(A) = \alpha$. If for any $M > 1$, there exists an instance I such that $A(I) > M \cdot \mathrm{OPT}(I)$, then we say that algorithm A has an unbounded worst-case performance ratio which implies that A is a poor algorithm.

All algorithms discussed in this paper use a level-by-level layer-by-layer packing scheme which is generalized from the strip (or shelf, level) packing scheme for rectangle packing. Coffman et al. [5] first studied two level-oriented packing algorithms in two dimensions, namely, Next-Fit-Decreasing-Height (NFDH) and First-Fit-Decreasing-Height (FFDH). It is shown that $r(\mathrm{NFDH}) = 2$ and $r(\mathrm{FFDH}) = 1.7$. Baker, Brown, and Katseff [2] investigated the best up-to-date strip packing algorithm for the rectangle packing problem with asymptotic performance bound 1.25. The strip packing strategy has also been used for the on-line rectangle packing problem [3]. The multidimensional packing problem has been studied in the setting of *vector packing* [7], [17] where pieces (i.e., rectangles or boxes) can only be packed in a corner-to-corner manner across a bin diagonally and the objective is to minimize the number of bins used. Since there is no geometric flavor, vector packing is different from bin packing in multi-dimensions. Karp, Luby, and Marchetti-Spaccamela [11] have investigated both multi-dimensional vector and bin packing problems from a probabilistic point of view. However, there is still a lack of combinatorial analysis results. A comprehensive survey on this active research area can be found in [6] where numerous solutions to all problems mentioned above (except the 3D packing problem) are summarized and compared.

The rest of the paper is organized as follows. In § 2, we show that algorithms NFDH and FFDH have unbounded worst-case performance ratios in three-dimensional packing. In § 3, algorithm G is developed and we show that $4.333 \leqq r(G) \leqq 4.571$. In § 4, the algorithm is refined and the asymptotic performance bound of the new algorithm is reduced to 3.25. Both algorithms are adapted for the case when all boxes have square bottoms and in this special case, their asymptotic performance bounds are 4 and 2.6875, respectively. All these algorithms are further refined to handle boxes with small bottoms.

2. Analysis of algorithms NFDH and FFDH. First we consider a simple packing algorithm called *level-by-level layer-by-layer* (LL^x) which is used by both algorithms NFDH and FFDH. For a list of boxes $L = (b_1, b_2, \cdots, b_n)$, algorithm LL^x packs boxes of L in the given order. It starts to pack b_1 from the lower-left hand corner on the bottom of box B. Then it fills the first *layer* from left to right until a box, b_i, cannot be packed into the current layer. At that time it packs b_i into the second layer. This process repeats until there is a box b_j such that it cannot be packed into the current layer and that there is not enough space to create a new layer in the first *level*. At that time algorithm LL^x starts to pack boxes in the first layer of the second level. A formal

description of algorithm LL^x can be found in Appendix B. Clearly the LL^x packing algorithm is simply a generalization of the level by level (or shelf) packing strategy in two-dimensional packing [3], [5].

Now given an instance I of the 3D packing problem, both algorithms NFDH and FFDH first sort I into a list $L=(b_{i_1}, b_{i_2}, \cdots, b_{i_n})$ such that $z_{i_1} \geqq z_{i_2} \geqq \cdots \geqq z_{i_n}$. Then NFDH applies the LL^x packing algorithm on L to generate a packing of I. Algorithm FFDH employs a variation of LL^x. When it packs a box b_i, FFDH always examines all layers one by one starting from the first layer of the first level. Whenever it finds a layer in which b_i can fit, b_i is put there. A new layer (and a new level if necessary) is created if no previous layer can accommodate b_i.

THEOREM 1. *For any integer $M>1$, there exists an instance I of the three-dimensional packing problem such that* $\mathrm{NFDH}(I) > M \cdot \mathrm{OPT}(I)$.

Proof. Let $I=\{b_1, b_2, \cdots, b_{2k}\}$ where

$$b_i = \begin{cases} \left(1, \dfrac{1}{k}, 1-(i-1)\varepsilon\right), & i=1,3,\cdots,2k-1, \\ \left(\dfrac{1}{k}, 1, 1-(i-1)\varepsilon\right), & i=2,4,\cdots,2k \end{cases}$$

and k is a large integer to be defined below. Clearly b_i's are already in decreasing order of height. Therefore after sorting, $L=(b_1, b_2, \cdots, b_{2k})$. It is easy to see that in the packing generated by NFDH, each box occupies a whole level. Hence

$$\mathrm{NFDH}(I) = \sum_{i=1}^{2k} (1-(i-1)\varepsilon) = 2k - k(2k-1)\varepsilon.$$

Now consider another packing in which we have two levels. The first level contains all b_i's where $i=1,3,\cdots,2k-1$ and the second level contains all b_i's where $i=2,4,\cdots,2k$. The height of this packing is $2-\varepsilon$. Thus we have

$$\mathrm{OPT}(I) \leqq 2-\varepsilon.$$

To ensure that

$$\frac{\mathrm{NFDH}(I)}{\mathrm{OPT}(I)} \geqq \frac{2k-k(2k-1)\varepsilon}{2-\varepsilon} > M,$$

we just need to have $k=M+1$ and $\varepsilon < 2/(2M^2+2M+1)$. □

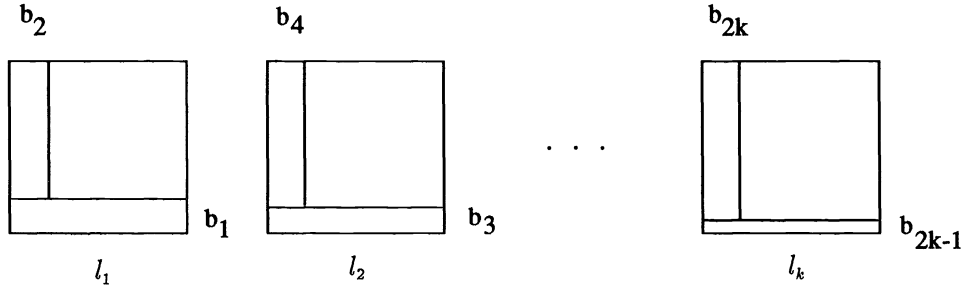
THEOREM 2. *For any integer $M>1$, there exists an instance I of the three-dimensional packing problem such that* $\mathrm{FFDH}(I) > M \cdot \mathrm{OPT}(I)$.

Proof. Consider a set of boxes $I=\{b_1, b_2, \cdots, b_{2k}\}$ where

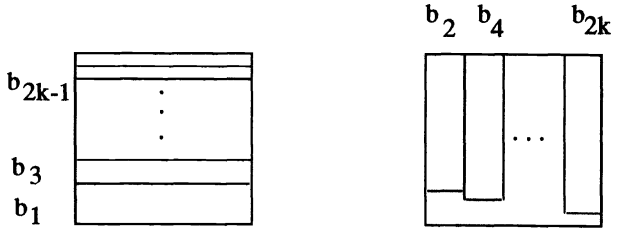
$$b_i = \begin{cases} \left(1, \dfrac{1}{k} - \dfrac{i-1}{2}\delta, 1-(i-1)\varepsilon\right), & i=1,3,\cdots,2k-1, \\ \left(\dfrac{1}{k}, 1-\dfrac{1}{k}+\dfrac{i-2}{2}\delta, 1-(i-1)\varepsilon\right), & i=2,4,\cdots,2k \end{cases}$$

where $\delta < 1/(k(k-1))$, k is a large integer, and ε is a small quantity to be defined. In the packing produced by algorithm FFDH, there are k levels $l_1, l_2, \cdots, l_k$ where l_i contains b_{2i-1} and b_{2i}, $1 \leqq i \leqq k$ (see Fig. 1(a) where the bird's-eye view of the k levels are shown). Hence

$$\mathrm{FFDH}(I) = \sum_{i=1}^{k} (1-2(i-1)\varepsilon) = k - k(k-1)\varepsilon.$$



(a) The FFDH packing



(b) A possible packing

FIG. 1. *The example in the proof of Theorem* 2.

However we can have the same two level packing (Fig. 1(b)) as in the proof of Theorem 1, i.e.,

$$\mathrm{OPT}(I) \leqq 2-\varepsilon.$$

Therefore to guarantee that

$$\frac{\mathrm{FFDH}(I)}{\mathrm{OPT}(I)} \geqq \frac{k-k(k-1)\varepsilon}{2-\varepsilon} > M,$$

let $k=2M+1$ and $\varepsilon < 1/(4M^2+M)$. □

Theorems 1 and 2 demonstrate that the performance of NFDH and FFDH is severely affected by simply using the LL^x packing strategy without refinement. The proofs of the two theorems are based on the following simple fact. Let (x, y) denote a rectangle of size x in the x-dimension and y in the y-dimension. Then the fact is that for any small $\varepsilon > 0$, the condition $\sum_{i=1}^{n} x_i y_i \leqq \varepsilon$ does not guarantee that the set of rectangles $\{(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)\}$ can be packed into a unit square. To see this, consider two rectangles $(\varepsilon/4, 1)$ and $(1, \varepsilon/4)$. Clearly the sum of their areas is $\varepsilon/2 < \varepsilon$, but they are not packable into a unit square.

A natural question to ask is, "would the performance of NFDH and FFDH be better if we put some restrictions on I?" For example, the bottom of any box b_i in I may satisfy $x_i \geqq y_i$ or may even be a square, i.e., $x_i = y_i$. We may also assume that the bottom of boxes in I are very small. For this purpose, we define $\Im_m$ ($m \geqq 1$ is an integer) to be the set of instances $I = \{b_1, b_2, \cdots, b_n\}$ such that $x_i = y_i \leqq 1/m$, $1 \leqq i \leqq n$.

Since the bottom shape is more regular and bottom sizes are small, it is hopeful that the amount of occupied area (utilization) in each level will be increased. Unfortunately this intuition is not correct.

THEOREM 3. *For any integers $M>1$ and $m\geqq 1$, there exists an instance $I\in\mathfrak{I}_m$ such that* NFDH $(I)>M\cdot$ OPT (I) *and* FFDH $(I)>M\cdot$ OPT (I).

Proof. We construct a set of boxes $I=\{b_1, b_2, \cdots, b_n\}$ as follows. Let $n=k^2(1+(k-1)^2)$. The bottom sizes are

$$(x_i, y_i)=\begin{cases}\left(\dfrac{1}{k}, \dfrac{1}{k}\right) & \text{if } i \bmod (1+(k-1)^2)=1,\\ \left(\dfrac{1}{k(k-1)}, \dfrac{1}{k(k-1)}\right) & \text{otherwise.}\end{cases}$$

These heights are designed such that $z_1>z_2>\cdots>z_n$ and, in particular,

$$z_{(i-1)n_1+1}=1-i\delta, \qquad 1\leq i\leq k$$

where $n_1=k(1+(k-1)^2)$ and $\delta=2/(k(k+1))$.

Now both NFDH and FFDH will produce the same packing in which there are k levels. In each level there are k layers and in each layer there are $1+(k-1)^2$ boxes. There is only one box with bottom size $(1/k, 1/k)$ in each layer at the left end. Note that the utilization of each level is only

$$k\left[\left(\frac{1}{k}\right)^2+(k-1)^2\left(\frac{1}{k(k-1)}\right)^2\right]=\frac{2}{k},$$

which can be arbitrarily small if k is large. Since the height of these k levels are $1-\delta, 1-2\delta, \cdots, 1-k\delta$, respectively, the height of the packing generated by either algorithm is

$$\text{NFDH}\,(I)=\text{FFDH}\,(I)=k-\frac{k(k+1)}{2}\,\delta=k-1.$$

Consider another packing in which all k^2 boxes with bottom size $(1/k, 1/k)$ are arranged in one level and all $k^2(k-1)^2$ boxes with bottom size $(1/k(k-1), 1/k(k-1))$ are arranged in another level. Clearly, the height of this packing is less than $2(1-\delta)$. Thus

$$\text{OPT}\,(I)<2-2\delta.$$

To ensure that $I\in\mathfrak{I}_m$ and

$$\frac{k-1}{2-2\delta}>\frac{k-1}{2}>M,$$

let $k>\max(2M+1, m)$. □

Note that layers may also be parallel to the y-dimension. Its corresponding algorithm is referred to as LL^y. Hence algorithm NFDH may invoke the LL^y packing procedure. In the sequel, we use NFDH^x and NFDH^y to denote the NFDH algorithm which calls LL^x and LL^y, respectively. If the superscript x or y is omitted, then NFDH stands for either NFDH^x or NFDH^y. Notations FFDH, FFDH^x, and FFDH^y are defined in a similar way. It is clear that Theorems 1, 2, and 3 are also valid for both algorithms NFDH^y and FFDH^y.

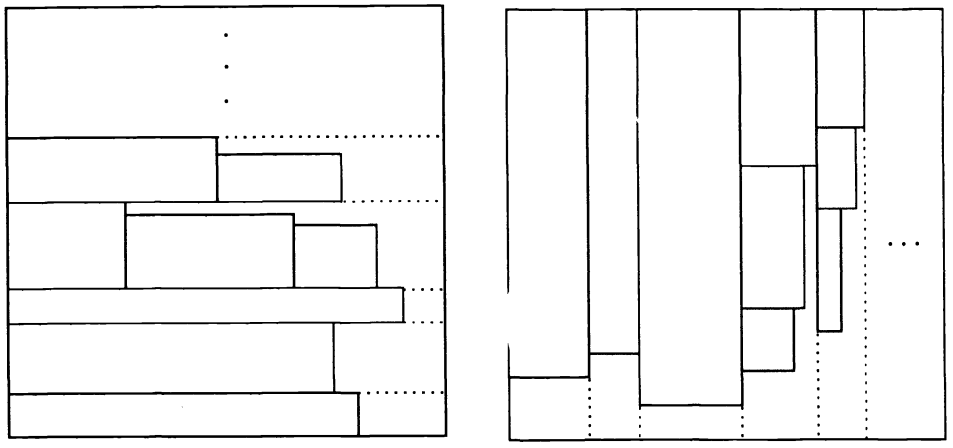
3. An algorithm and its analysis. The main problem for algorithms NFDH and FFDH is that the utilization in each level cannot be guaranteed to be above a certain percentage. However under certain conditions, we can construct packing strategies which have guaranteed utilization. In this section, we develop an algorithm to solve the 3D packing problem. Similar to algorithms NFDH and FFDH, our algorithm also adopts the nonincreasing height heuristic in the z-dimension. Packings are also arranged in the level-by-level layer-by-layer fashion. However the LL^x (LL^y) packing strategy is substantially refined.

First, let us consider a packing algorithm called L^x. Algorithm L^x is to pack a given list of rectangles $L=((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n))$ into a unit square. It first picks out those rectangles in L with width larger than $\frac{1}{2}$ and put each of them in one layer. Then it packs the remaining rectangles in the given order layer by layer using the next-fit strategy, i.e., whenever a rectangle cannot fit in the current layer, a new layer is created and all previous layers are no longer used. Figure 2(a) illustrates an L^x packing. Note that layers are parallel to the x-dimension, however, they may also be parallel to the y-dimension. We call the corresponding packing algorithm L^y (see Fig. 2(b)). Lemma 1 proved in [15] gives a sufficient condition which guarantees the packability of a list of rectangles by the L^x packing algorithm (similarly for L^y).

LEMMA 1 [15]. *A list of rectangles* $L=((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n))$ *can be packed into a unit square by using the* L^x *packing algorithm if for all* i, $1 \leqq i \leqq n$: $x_i \geqq y_i$, $y_1 \geqq y_2 \geqq \cdots \geqq y_n$ *and* $\sum_{i=1}^{n} x_i y_i \leqq \frac{7}{16}$. □

Algorithm L^x *may* fail to pack L into a unit square if one or more of the three conditions in Lemma 1 is violated, i.e., all conditions are necessary to guarantee the packability of L. To show that the shape of each rectangle in L is restricted to satisfy $x_i \geqq y_i$, consider a list L of two rectangles $(\varepsilon/4, 1)$ and $(1, \varepsilon/4)$ with $\varepsilon < \frac{7}{8}$. It satisfies the last two conditions, but L is not packable by L^x because the first condition is not met. To see that the list L should be in nonincreasing order of depth, consider $L=((\frac{1}{3}, \varepsilon), (\frac{1}{3}, \varepsilon), (\frac{1}{3}, \frac{1}{3}), (\frac{1}{3}, \varepsilon), (\frac{1}{3}, \varepsilon), (\frac{1}{3}, \frac{1}{3}), (\frac{1}{3}, \varepsilon), (\frac{1}{3}, \varepsilon), (\frac{1}{3}, \frac{1}{3}), (\frac{1}{3}, \varepsilon))$ where $0 < \varepsilon \leqq \frac{5}{112}$. Clearly, each rectangle in L satisfies $x \geqq y$. The sum of their area is $\frac{7}{3}\varepsilon + \frac{1}{3}$ which is less than or equal to $\frac{7}{16}$. However L is not packable by using L^x because L is not in nonincreasing order of y. Finally, to show that the cumulative area of all rectangles in L should be no more than $\frac{7}{16}$, consider $L=((\frac{1}{2}+\delta, \frac{1}{4}+\delta), (\frac{1}{2}+\delta, \frac{1}{4}+\delta), (\frac{1}{2}+\delta, \frac{1}{4}+\delta), (\frac{1}{4}-2\delta, \frac{1}{4}-2\delta))$. It is easy to check that all rectangles in L satisfy $x \geqq y$ and that L is in nonincreasing order of depth. The cumulative area of the four rectangles in L is $\frac{7}{16}+\frac{5}{4}\delta+7\delta^2$. Thus for any $\varepsilon > 0$, we can always choose $\delta > 0$ such that $\frac{5}{4}\delta + 7\delta^2 \leqq \varepsilon$. However, L^x cannot pack L into a unit square and this demonstrates that the bound $\frac{7}{16}$ is tight.

Based on Lemma 1, we design the following algorithm called G to solve the 3D packing problem. A preliminary version of this algorithm is reported in [13] where the algorithm is designed with the assumption that for all i, $1 \leqq i \leqq n$: $x_i \geqq y_i$. However this restriction is removed here. Algorithm 1 shows a complete description of G. Given an instance I, algorithm G first splits I into two subsets I_x and I_y according to whether $x_i \geqq y_i$ or not (Step 1). Then the packings of I_x and I_y are generated separately (Steps 2 and 3), and finally their packings are combined as a packing of I (Step 4). To pack I_x, algorithm G also arranges boxes of I_x in nonincreasing order of height (Step 2.2) and packs boxes level by level (Step 2.5). To avoid those situations in the proofs of Theorems 1-3 to occur, algorithm G uses additional heuristics in the x- and y-dimensions (Steps 1, 2.1, 2.3, and 2.4), which guarantee a minimum utilization ($\frac{7}{32}$) in each level (except in the last level). Note that by Lemma 1, Steps 1, 2.3, and 2.4 guarantee that in Step 2.5, L^x can pack L_i ($1 \leqq i \leqq v$) layer by layer in only one level. The packing of I_y is obtained in a similar way using L^y.



(a) An L^x packing (b) An L^y packing

FIG. 2. *Illustration of the* L^x *and the* L^y *packings.*

ALGORITHM *G.*

1. Divide I into two subsets $I_x=\{b_i \mid x_i \geqq y_i\}$ and $I_y=\{b_i \mid y_i > x_i\}$. Without loss of generality, let
$$I_x=\{b_1, b_2, \cdots, b_m\} \quad \text{and} \quad I_y=\{b_{m+1}, b_{m+2}, \cdots, b_n\}.$$
2. Generate the packing of I_x as follows:
   2.1. Choose those b_i's in I_x with $x_i y_i > \frac{7}{32}$. Without loss of generality, let these b_i's be $b_{p+1}, b_{p+2}, \cdots, b_m$.
   2.2. Sort the set $\{b_1, b_2, \cdots, b_p\}$ into a list L in nonincreasing order of height. Assume that $L=(b_1, b_2, \cdots, b_p)$.
   2.3. Divide the list L into v sublists $L_1, L_2, \cdots, L_v$ where
   $$L_i=(b_{k_{i-1}+1}, b_{k_{i-1}+2}, \cdots, b_{k_i}), \qquad 1 \leqq i \leqq v$$
   and $k_0=0$, $k_v=p$ such that for all i, $1 \leqq i \leqq v-1$, $\sum_{i=k_{v-1}+1}^{k_v} x_i y_i \leqq \frac{7}{16}$ but $\sum_{i=k_{v-1}+1}^{k_v+1} x_i y_i > \frac{7}{16}$.
   2.4. Sort L_i $(1 \leqq i \leqq v)$ in nonincreasing order of depth.
   2.5. Construct the packing of I_x as follows. Pack b_i $(p+1 \leqq i \leqq m)$ into a level by itself. For each list L_i $(1 \leqq i \leqq v)$, apply L^x to produce a packing of L_i in only one level. Combine the above $m-p+v$ levels to give a packing of I_x.
3. Generate the packing of I_y in a similar way as Step 2 except that L_i is sorted in nonincreasing order of width and its packing is produced by L^y.
4. Concatenate the packings of I_x and I_y to give a packing of I.

ALGORITHM 1

THEOREM 4. *For any instance I of the three-dimensional packing problem in which no box has height exceeding Z, we have*

$$G(I) < 4\tfrac{4}{7} \cdot \mathrm{OPT}(I) + 2Z.$$

Proof. Consider the packing of I_x generated in Step 2.5. Let h_i be the height of the level for L_i, $1 \leqq i \leqq v$, i.e., $h_i = \max_{k_{i-1}+1 \leqq j \leqq k_i}(z_j)$. Since box b_i $(p+1 \leqq i \leqq m)$ is itself a level, the height of the packing of I_x is

$$H(I_x)=(z_{p+1}+\cdots+z_m)+(h_1+h_2+\cdots+h_v). \tag{1}$$

Let $V(S)=\sum_{(x,y,z)\in S} xyz$. Then $V(I)=V(I_x)+V(I_y)$. Consider $V(I_x)$,

$$V(I_x)=\sum_{i=p+1}^{m} x_i y_i z_i+\sum_{i=1}^{v} V(L_i)$$

where $V(L_i)=\sum_{j=k_{i-1}+1}^{k_i} x_j y_j z_j$. Because $x_i y_i > \frac{7}{32}$ for $p+1\leqq i\leqq m$ (Step 2.1),

$$\sum_{i=p+1}^{m} x_i y_i z_i > \frac{7}{32}(z_{p+1}+\cdots+z_m). \tag{2}$$

In the level for L_i, $1\leqq i\leqq v-1$, we observe that the occupied area is larger than $\frac{7}{32}$, i.e.,

$$\sum_{j=k_{i-1}+1}^{k_i} x_j y_j > \tfrac{7}{32},$$

otherwise box b_{k_i+1} could be added to L_i in Step 2.3. Also note that each box in L_i has a height no less than h_{i+1} (Step 2.2), i.e., $z_j\geqq h_{i+1}$, $k_{i-1}+1\leqq j\leqq k_i$. Hence

$$V(L_i)>\tfrac{7}{32}h_{i+1}, \qquad 1\leqq i\leqq v-1. \tag{3}$$

Combining (2) and (3), we have

$$V(I_x)>\tfrac{7}{32}(z_{p+1}+\cdots+z_m+h_2+\cdots+h_v),$$

which together with (1) implies that

$$H(I_x)<\tfrac{32}{7}V(I_x)+h_1\leqq\tfrac{32}{7}V(I_x)+Z. \tag{4}$$

Similarly for Step 3,

$$H(I_y)<\tfrac{32}{7}V(I_y)+Z. \tag{5}$$

Thus from (4) and (5), the height of the packing of I generated by algorithm G (Step 4) is

$$G(I)=H(I_x)+H(I_y)<\tfrac{32}{7}(V(I_x)+V(I_y))+2Z.$$

Since $\mathrm{OPT}(I)\geqq V(I)=V(I_x)+V(I_y)$, the theorem follows. □

Theorem 4 shows that $4\frac{4}{7}\approx 4.571$ is an asymptotic performance bound of algorithm G. However we are unable to show that it is tight nor prove another bound which is less than $4\frac{4}{7}$. Next, we give a lower bound for $r(G)$.

COROLLARY. *There exists an instance I such that $G(I)=4\frac{1}{3}\cdot\mathrm{OPT}(I)$. Hence by Theorem 4, we have $4\frac{1}{3}\leqq r(G)\leqq 4\frac{4}{7}$.*

Proof. Consider a set of boxes, $I=I_1\cup I_2$, where I_1 contains $4n$ boxes of size $(1,\frac{7}{32}+\delta,1)$, I_2 contains n boxes of size $(1,\frac{1}{8}-4\delta,1)$, $\delta<\frac{1}{256}$ and $n \bmod 3=0$. In the packing generated by algorithm G, each box in I_1 occupies a level. Boxes in I_2 are divided into $n/3$ groups each having 3 boxes. Since each level has height 1, we have $G(I)=4n+n/3=4\frac{1}{3}n$. In the optimal packing, there are n levels each containing four boxes from I_1 and one box from I_2. Hence, $\mathrm{OPT}(I)=n$. The result then follows. □

In Theorem 3, we demonstrate that algorithms NFDH and FFDH perform poorly even for very restricted cases. In the following we show an opposite of Theorem 3. First we have the following lemma where a square (x,x) is represented as x.

LEMMA 2. *A list of squares $L=(x_1,x_2,\cdots,x_n)$ can be packed into a unit square by using the L^x procedure if $x_1\geqq x_2\geqq\cdots\geqq x_n$ and $\sum_{i=1}^{n} x_i^2\leqq x_1^2+(1-x_1)^2$. Hence if $x_1\leqq 1/m$ where m is an integer and $m\geqq 2$, then the condition*

$$\sum_{i=1}^{n} x_i^2\leqq\frac{1}{m^2}+\left(1-\frac{1}{m}\right)^2$$

guarantees that L^x *can pack* L *into a unit square. If* $x_1 \leqq 1$, *then the condition*

$$\sum_{i=1}^{n} x_i^2 \leqq \frac{1}{2}$$

guarantees that L^x *can pack* L *into a unit square. The bound* $\frac{1}{2}$ *is tight.*

Proof. The proof of the first result can be found in [15]. Since $x_1 \leqq 1/m$ $(m \geqq 1)$ implies that

$$x_1^2+(1-x_1)^2 = 2\left(x_1-\frac{1}{2}\right)^2+\frac{1}{2} \geqq \begin{cases} \frac{1}{2} & \text{if } m=1, \\ 2\left(\frac{1}{m}-\frac{1}{2}\right)^2+\frac{1}{2} & \text{if } m \geqq 2 \end{cases}$$

we have the other consequences. To show the tightness of $\frac{1}{2}$, consider two squares of size $\frac{1}{2}$ and $\frac{1}{2}+\delta$. They cannot be packed into a unit square in any way but the sum of their areas is $\frac{1}{2}+\delta+\delta^2$ which can be arbitrarily close to $\frac{1}{2}$ as $\delta \to 0$. □

Since instances are restricted to be of a certain form, algorithm G can be simplified. We define algorithm G_m $(m \geqq 1)$ as follows. Given an instance $I \in \mathfrak{I}_1$, algorithm G_1 only performs Steps 2.1–2.5 of algorithm G and the quantities $\frac{7}{32}$ and $\frac{7}{16}$ in Steps 2.1 and 2.3 are changed to $\frac{1}{4}$ and $\frac{1}{2}$, respectively (by Lemma 2). Given an instance $I \in \mathfrak{I}_m$ $(m \geqq 2)$, algorithm G_m only does Steps 2.2–2.5 of algorithm G. The bound $\frac{7}{16}$ is changed to $1/m^2+(1-1/m)^2$ (by Lemma 2) and only v levels are generated in Step 2.5 (since there is no Step 2.1).

THEOREM 5. *For any* $I \in \mathfrak{I}_m$ $(m \geqq 1)$ *where no box has height exceeding* Z, *we have*

$$G_m(I) < \alpha_m \cdot \mathrm{OPT}(I) + Z$$

where

$$a_m = \begin{cases} 4 & \text{if } m=1, \\ \left(\frac{m}{m-1}\right)^2 & \text{if } m \geqq 2. \end{cases}$$

Furthermore, the bound α_m *is tight for all* $m \geqq 1$, *i.e.,* $r(G_m) = \alpha_m$.

Proof. The proof that α_m is an asymptotic performance bound of algorithm G_m is similar to the proof of Theorem 4 and is left to the interested reader.

In the following, we will construct an instance I_m to show that α_m is tight. Since $\alpha_1 = \alpha_2$, the instance I_2 constructed below can also be used for α_1. Hence let $m \geqq 2$.

Consider $I_m = \{b_1, b_2, \cdots, b_n\}$ where $n = k^2m^2[1+(m-1)^2]$, $k \geqq m$ and

$$b_i = \begin{cases} \left(\frac{1}{k}, \frac{1}{k}, 1-(i-1)\xi\right) & \text{if } i \bmod (1+(m-1)^2) = 0, \\ \left(\frac{1}{m}, \frac{1}{m}, 1-(i-1)\xi\right) & \text{otherwise.} \end{cases}$$

Additional requirements on the large integer k and the small positive quantity ξ are to be described below. Clearly, there are basically two kinds of boxes. For ease of reference, we call those boxes with bottom $(1/m, 1/m)$ as X-boxes and those with bottom $(1/k, 1/k)$ as Y-boxes.

Now let us consider the packing of I_m generated by algorithm G_m. After sorting in nonincreasing order of height (Step 2.2), we get the list L which is exactly in the order $b_1, b_2, \cdots, b_n$. In Step 2.3, G_m divides the list L into $v = k^2m^2$ groups and each

group contains $(m-1)^2$ X-boxes and one Y-box as illustrated below:

$$L=(\underbrace{\underbrace{X, X, \cdots, X, Y}_{g=1+(m-1)^2}, \underbrace{X, X, \cdots, X, Y}_{g=1+(m-1)^2}, \cdots, \underbrace{X, X, \cdots, X, Y}_{g=1+(m-1)^2}}_{v=k^2m^2 \text{ groups}}).$$

This is because the cumulative bottom area of g X-boxes and 1 Y-box exceeds the bound $1/m^2+(1-1/m)^2$ in Step 2.3. Thus there are v levels in the packing of I_m produced by G_m and the height of the ith level is $z_{(i-1)g+1}=1-(i-1)g\xi$. Hence we have

$$G_m(I_m)=z_1+z_{g+1}+z_{2g+1}+\cdots+z_{(v-1)g+1}=v-\frac{v(v-1)}{2}\cdot g\xi. \tag{6}$$

Let us consider another packing in which there are

$$\frac{v(m-1)^2}{m^2}=k^2(m-1)^2$$

levels each having m^2 X-boxes, plus $v/k^2=m^2$ levels each having k^2 Y-boxes. Since the height of each level is less than one, we have

$$\mathrm{OPT}(I_m)<k^2(m-1)^2+m^2. \tag{7}$$

Combining (6) and (7) gives

$$\lim_{\xi\to 0}\frac{G_m(I_m)}{\mathrm{OPT}(I_m)}=\frac{k^2m^2}{k^2(m-1)^2+m^2}.$$

Clearly, the above ratio is less than α_m, but it can be made arbitrarily close to α_m as $k\to\infty$. □

Note that $\alpha_1=\alpha_2>\alpha_3>\cdots>\alpha_m$ and $\lim_{m\to\infty}\alpha_m=1$, i.e., as m gets larger, the performance of G_m becomes better and better and close to optimal. Obviously algorithms G_m's can be generalized to handle boxes with rectangular bottoms. Let $\Im_m^*$ denote the set of instances of the 3D packing problem in which $x_i\leqq 1/m$ and $y_i\leqq 1/m$ but x_i need not be equal to y_i, $1\leqq i\leqq n$. The generalization is based on the following lemma which is proved in [13].

LEMMA 3. *A list of rectangles* $L=((x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n))$ *can be packed into a unit square by* L^x *if for all* $1\leqq i\leqq n$: $y_i\leqq x_i\leqq 1/m$ $(m\geqq 3)$, $y_1\geqq y_2\geqq\cdots\geqq y_n$ *and* $\sum_{i=1}^n x_iy_i\leqq(1-1/m)^2$. □

Define algorithm G_m^* $(m\geqq 3)$ as follows. G_m^* does each step in algorithm G except Step 2.1. The bound $\frac{7}{16}$ in Step 2.3 is changed to $(1-1/m)^2$.

THEOREM 6. *For any* $I\in\Im_m^*$ $(m\geqq 3)$ *where no box has height exceeding* Z, *we have*

$$G_m^*(I)<\beta_m\cdot\mathrm{OPT}(I)+2Z$$

where $\beta_m=m/(m-2)$ *and* β_m *is a tight bound.*

Proof. The proof for the inequality is similar to that of Theorems 4 and 5. To show the tightness of β_m, we construct an instance I_m^* similar to I_m in the proof of Theorem 5. Let $I_m^*=\{b_1, b_2, \cdots, b_n\}$ where $n=k^2m^2(m-1)^2$, $k\geqq m$ and

$$b_i=\begin{cases}\left(\frac{1}{k}, \frac{1}{k}, 1-(i-1)\xi\right) & \text{if } i \bmod (m-1)^2=0,\\ \left(\frac{1}{m}, \frac{1}{m}, 1-(i-1)\xi\right) & \text{otherwise.}\end{cases}$$

Then we can enforce I_m^* to be divided into k^2m^2 groups and each group contains $(m-1)^2-1$ X-boxes and 1 Y-box. We leave further verification to the interested reader. □

4. An improved algorithm. Theorems 5 and 6 show that algorithm G works quite well when boxes have small bottoms. However in general, the asymptotic performance bounds 4.571 and 4 are still too large. We observe that the upper bounds $\frac{7}{16}$ and $\frac{1}{2}$ are only used to guarantee that each sublist can be packed into the unit square (Step 2.3). They are sufficient but not necessary conditions, and thus severely affect the performance of G. However both $\frac{7}{16}$ and $\frac{1}{2}$ cannot be increased if L^x is used.

In this section, we analyze an algorithm called C which also packs boxes level by level like algorithm G but has more refined heuristics in the x- and y-dimensions. It is based on the observation that when boxes have similar bottoms, even NFDH and its variants can generate good packings. We have seen in § 2 that algorithm NFDH is not able to find good packings even for $I \in \Im_m^*$ where m can be very large. However, if we put more restrictions on I to increase the similarity among boxes, then NFDH (with slight modification) may yield good packings. For $m \geqq 2$, let

$$\Im_m^x = \left\{ I = \{b_1, b_2, \cdots, b_n\} \,|\, I \in \Im_m^* \text{ and } \forall 1 \leqq i \leqq n: \frac{1}{m+1} < x_i \leqq \frac{1}{m} \right\},$$

$$\Im_m^y = \left\{ I = \{b_1, b_2, \cdots, b_n\} \,|\, I \in \Im_m^* \text{ and } \forall 1 \leqq i \leqq n: \frac{1}{m+1} < y_i \leqq \frac{1}{m} \right\}.$$

Given an instance $I \in \Im_m^x$, we define algorithm NFDH_m^x as follows. Initially, NFDH_m^x sorts I into a list L in nonincreasing order of height. Then it invokes the LL^y (not LL^x!) packing procedure to generate a packing of I. Algorithm NFDH_m^y is defined in a similar way with the differences that it accepts input from $\Im_m^y$ and invokes LL^x.

LEMMA 4. *In the packing of $I \in \Im_m^x$ generated by algorithm NFDH_m^x, the utilization of each level except the last level is larger than $(m-1)/(m+1)$. The result is also true in the packing of $I \in \Im_m^y$ generated by NFDH_m^y.*

Proof. Let $I \in \Im_m^x$. Suppose there are v levels $l_1, l_2, \cdots, l_v$ in the NFDH_m^x packing. Consider level l_i $(1 \leqq i \leqq v-1)$. Since $1/(m+1) < x_i \leqq 1/m$, there are exactly m layers parallel to the y-dimension (see Fig. 3 for a bird's-eye view of a level). Since no box has a depth exceeding $1/m$, the depth to which a layer is filled is larger than $(1-1/m)$; otherwise NFDH_m^x will pack another box in that layer. Because each box has a width larger $1/(m+1)$, we know that the utilization of level l_i is larger than

$$m \cdot \frac{1}{m+1} \cdot \left(1 - \frac{1}{m}\right) = \frac{m-1}{m+1}.$$

The proof for NFDH_m^y is similar. □

Note that algorithm NFDH_m^x (NFDH_m^y) should use LL^y (LL^x). If NFDH_m^x (or NFDH_m^y) uses LL^x (LL^y), then we can easily construct counterexamples to demonstrate that Lemma 4 is no longer valid. We leave this verification to the reader.

The main strategy of algorithm C is to separate boxes with different bottoms in a way more refined than Step 2.1 of Algorithm G so that the utilization in each level can be improved. If we view the bottom size of a box as a point (x, y) in the region $(0, 1] \times (0, 1]$, then Algorithm G distinguishes four cases according to bottom shape and size (Fig. 4(a)). However, Algorithm C splits I into eight subsets (Fig. 4(b)), then generates their packings individually, and finally combines the eight packings. A complete description of C is given in Algorithm 2 which is self-explanatory.

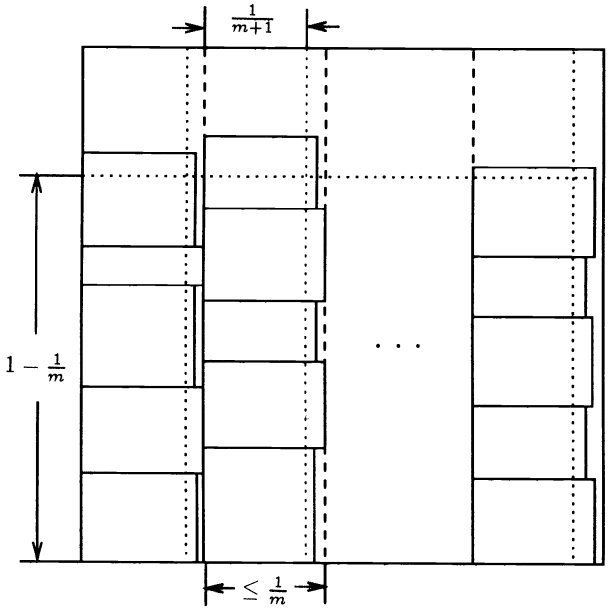


FIG. 3. *A level in an* NFDH_m^x *packing.*

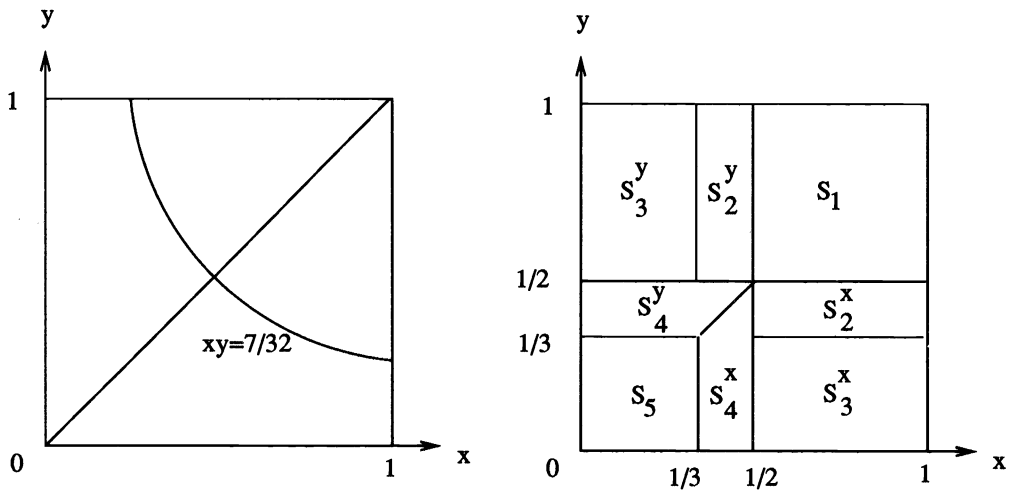


FIG. 4. *Partition of input in Algorithms G and C.*

ALGORITHM *C.*

1. Partition the set I into eight subsets:

$$S_1=\{b_i \mid x_i>\tfrac{1}{2} \text{ and } y_i>\tfrac{1}{2}\};$$

$$S_2^x=\{b_i \mid x_i>\tfrac{1}{2} \text{ and } \tfrac{1}{3}<y_i\leqq\tfrac{1}{2}\}; \qquad S_2^y=\{b_i \mid y_i>\tfrac{1}{2} \text{ and } \tfrac{1}{3}<x_i\leqq\tfrac{1}{2}\};$$

$$S_3^x=\{b_i \mid x_i>\tfrac{1}{2} \text{ and } y_i\leqq\tfrac{1}{3}\}; \qquad S_3^y=\{b_i \mid y_i>\tfrac{1}{2} \text{ and } x_i\leqq\tfrac{1}{3}\};$$

$$S_4^x=\{b_i \mid \tfrac{1}{3}<x_i\leqq\tfrac{1}{2} \text{ and } y_i\leqq x_i\}; \qquad S_4^y=\{b_i \mid \tfrac{1}{3}<y_i\leqq\tfrac{1}{2} \text{ and } x_i<y_i\};$$

$$S_5=\{b_i \mid x_i\leqq\tfrac{1}{3} \text{ and } y_i\leqq\tfrac{1}{3}\}$$

2. Generate the packing of S_1 such that each level contains one box only.
3. Generate the packing of S_i^d by applying NFDH^d where $i=2, 3$ and $d=x, y$.
4. Generate the packing of S_4^d by using NFDH$_2^d$ for $d=x, y$.
5. Generate the packing of S_5 by using G_3^*.
6. Combine the eight packings obtained in Steps 2–5 to form a packing of I.

ALGORITHM 2

THEOREM 7. *For any instance I of the three-dimensional packing problem in which no box has height exceeding Z, we have*

$$C(I) \leqq 3.25 \cdot \mathrm{OPT}(I) + 8Z.$$

Moreover, the bound 3.25 is tight.

Proof. Consider the packing of I produced by Algorithm C. Let h_1 denote the height of the packing of S_1. According to Step 2,

$$h_1 = \sum_{b_i=(x_i,y_i,z_i)\in S_1} z_i.$$

Let $h_i^d (i=2, 3, 4, 5;\ d=x, y)$ be the height of the packing of S_i^d minus the height of the *first* level in that packing. (Note that Algorithm G_3^* divides S_5 into two subsets S_5^x and S_5^y, and generates their packings separately.) If the height of any box is no more than Z, then

$$C(I) \leqq h_1 + H + 8Z \tag{8}$$

where $H = \sum_{i=2}^{5} (h_i^x + h_i^y)$. Let $V(S) = \sum_{b_i\in S} x_i y_i z_i$. Clearly,

$$V(S_1) > \tfrac{1}{4} h_1.$$

Now consider the packing of S_2^x generated in Step 3. Suppose there are ℓ levels $l_1, l_2, \cdots, l_\ell$. Let h_i^{2x} denote the height of l_i and V_i be the total volume of all boxes in l_i, $1 \leqq i \leqq \ell$. Since the utilization of level l_i $(1 \leqq i \leqq \ell - 1)$ is larger than $\frac{1}{3}$ (note that there are exactly two boxes in l_i) and the height of each box in l_i is at least h_{i+1}^{2x}, we have

$$V(S_2^x) > V_1 + V_2 + \cdots + V_{\ell-1} > \tfrac{1}{3}(h_2^{2x} + h_3^{2x} + \cdots + h_\ell^{2x}) = \tfrac{1}{3} h_2^x.$$

The above argument can also be applied to S_2^y. The reader may readily give similar arguments for S_3^x, S_3^y (trivial), S_4^x, S_4^y (by Lemma 4 since $S_4^x \in \Im_2^x$, $S_4^y \in \Im_2^y$) and S_5^x, S_5^y (by Lemma 3). So we have

$$V(S_i^d) > \tfrac{1}{3} h_i^d, \qquad i=2, 3, 4, 5, \quad d=x, y.$$

Therefore,

$$\mathrm{OPT}(I) \geqq V(I) = V(S_1) + \sum_{i=2}^{5} (V(S_i^x) + V(S_i^y)) > \frac{h_1}{4} + \frac{H}{3}.$$

Since we also have $\mathrm{OPT}(I) \geqq h_1$,

$$\mathrm{OPT}(I) \geqq \max\left(h_1, \frac{h_1}{4} + \frac{H}{3}\right). \tag{9}$$

Now let us examine the ratio

$$R = \frac{h_1 + H}{\max(h_1, \frac{1}{4}h_1 + \frac{1}{3}H)}. \tag{10}$$

If $h_1 \geqq \frac{1}{4}h_1 + \frac{1}{3}H$, i.e., $h_1 \geqq \frac{4}{9}H$, then

$$R = \frac{h_1 + H}{h_1} = 1 + \frac{H}{h_1} \leqq 1 + \frac{9}{4} = 3.25.$$

If $h_1 \leqq \frac{1}{4}h_1 + \frac{1}{3}H$, i.e., $h_1 \leqq \frac{4}{9}H$, then

$$R = \frac{h_1 + H}{\frac{1}{4}h_1 + \frac{1}{3}H}.$$

In this case, it is easy to check that R is a strictly increasing function of h_1. Hence when $h_1 = \frac{4}{9}H$, R gets its maximum value which is 3.25. In a word, we always have

$$R \leqq 3.25. \tag{11}$$

By combining (8)-(11), we know that for any instance I of the 3D packing problem in which no box has height exceeding Z, we have

$$C(I) \leqq 3.25 \cdot \mathrm{OPT}(I) + 8Z,$$

which implies that 3.25 is an asymptotic performance bound of algorithm C.

To complete the proof of the theorem, we need to demonstrate that the bound 3.25 is tight. Consider a set of boxes $I = I_1 \cup I_2$. There are $4n$ boxes of size $(\frac{1}{2}+\delta, \frac{1}{2}+\delta, 1)$ in I_1 and $90n$ boxes in I_2, i.e., $I_2 = \{b_1, b_2, \cdots, b_{90n}\}$ where $k \geqq 3$ and

$$b_i = \begin{cases} \left(\frac{1}{3}, \frac{1}{3}, 1-(\mathrm{i}-1)\xi\right) & \text{if } i \bmod 10 = 1, \\ \left(\frac{1}{6}-\delta, \frac{1}{6}-\delta, 1-(i-1)\xi\right) & \text{if } i \bmod 10 = 2, 3, \cdots, 9, \\ \left(\frac{1}{k}, \frac{1}{k}, 1-(i-1)\xi\right) & \text{if } i \bmod 10 = 0. \end{cases}$$

When Algorithm C is applied to I, $S_1 = I_1$, $S_5 = I_2$ and all the other subsets are empty. It is easy to see that the height of the packing of I_1 is $4n$. Now let us consider the packing of I_2 generated by algorithm G_3^*. First it sorts I_2 into a list L in nonincreasing order of height. Clearly, $L = (b_1, b_2, \cdots, b_{90n})$. Then G_3^* divides L into sublists such that the cumulative bottom area of all boxes in each sublist does not exceed $\frac{4}{9}$ (cf. Lemma 3). If

$$\frac{1}{k^2} - \frac{8}{3}\delta + 8\delta^2 > 0,$$

then it is not hard to verify that there are $9n$ sublists $L_1, L_2, \cdots, L_{9n}$ each having 10 boxes, i.e.,

$$L_i = (b_{10(i-1)+1}, b_{10(i-1)+2}, \cdots, b_{10i}), \qquad 1 \leqq i \leqq 9n.$$

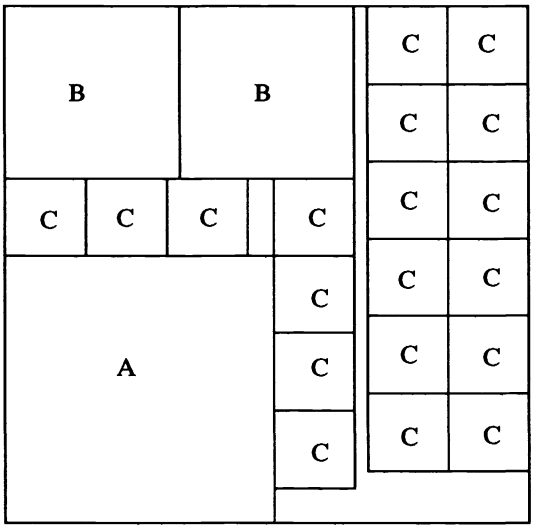
Thus the height of the packing of I_2 is

$$\sum_{i=1}^{9n} z_{10(i-1)+1} = \sum_{i=1}^{9n} (1 - 10(i-1)\xi) = 9n - 45n(9n-1)\xi.$$

Therefore,

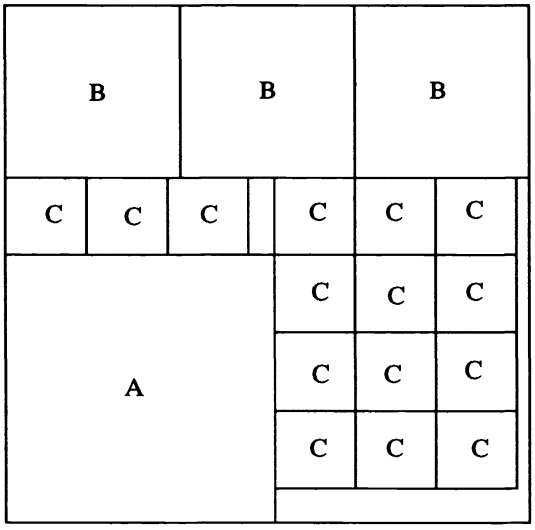
$$C(I) = 13n - 45n(9n-1)\xi.$$

Let us consider another packing in which there are $4n+1$ levels. Among them, $3n$ levels are of the same type whose bird's-eye view is illustrated in Fig. 5(a) and n levels



A: $(\frac{1}{2}+\delta, \frac{1}{2}+\delta)$
B: $(\frac{1}{3}, \frac{1}{3})$
C: $(\frac{1}{6}-\delta, \frac{1}{6}-\delta)$

(a) 3n levels of this type



(b) n levels of this type

FIG. 5. *Illustration of a possible packing (Theorem 7).*

are of another type whose bird's-eye view is illustrated in Fig. 5(b). Note that we can always let k be sufficiently large such that all boxes with bottom $(1/k, 1/k)$ are packed into one level. When $\delta \to 0$ and $\xi \to 0$, this is actually an optimal packing. Hence

$$\lim_{\delta,\xi\to 0} \frac{C(I)}{\mathrm{OPT}(I)} = \frac{13n}{4n+1} < 3.25.$$

The above ratio can be arbitrarily close to 3.25 as $n \to \infty$. □

When Algorithm C is applied to an instance $I \in \mathfrak{I}_1$, i.e., all boxes in I have square bottoms, there are at most three nonempty subsets obtained in Step 1, namely, S_1, S_4^x, and S_5. Also note that the packing of S_5 can be generated by using Algorithm G_3 instead of G_3^*. Thus Algorithm C can be simplified to C_1 as described in Algorithm 3 and we demonstrate in the next theorem that Algorithm C_1 has better performance.

ALGORITHM C_1.

1. Partition the set I into three subsets:

$$S_1 = \{b_i = (x_i, x_i, z_i) \mid x_i > \tfrac{1}{2}\};$$

$$S_2 = \{b_i = (x_i, x_i, z_i) \mid \tfrac{1}{3} < x_i \leqq \tfrac{1}{2}\};$$

$$S_3 = \{b_i = (x_i, x_i, z_i) \mid x_i \leqq \tfrac{1}{3}\}.$$

2. Generate the packing of S_1 with one box per level; generate the packing of S_2 using NFDH^x; generate the packing of S_3 using G_3.
3. Combine the packings of S_i, $i = 1, 2, 3$ to give a packing of I.

ALGORITHM 3

THEOREM 8. *For any instance $I \in \mathfrak{I}_1$ in which no box has height exceeding Z, we have*

$$C_1(I) \leqq 2.6875 \cdot \text{OPT}(I) + 2Z.$$

Furthermore, the bound 2.6875 *is tight.*

Proof. The proof is similar to that of Theorem 7. We give an outline of the proof as follows and leave detailed validation to the reader. Note that in the packing of S_2 (and S_3 also), the utilization in each level except the last one is larger than $\frac{4}{9}$. Hence the ratio R is

$$R = \frac{h_1 + H}{\max(h_1, \frac{1}{4}h_1 + \frac{4}{9}H)} \leqq 2\frac{11}{16} = 2.6875.$$

To show the tightness of 2.6875, we construct a set of boxes $I = I_1 \cup I_2$. There are $16n$ boxes of size $(\frac{1}{2}+\delta, \frac{1}{2}+\delta, 1)$ in I_1 and $378n$ boxes in I_2, i.e., $I_2 = \{b_1, b_2, \cdots, b_{378n}\}$ where

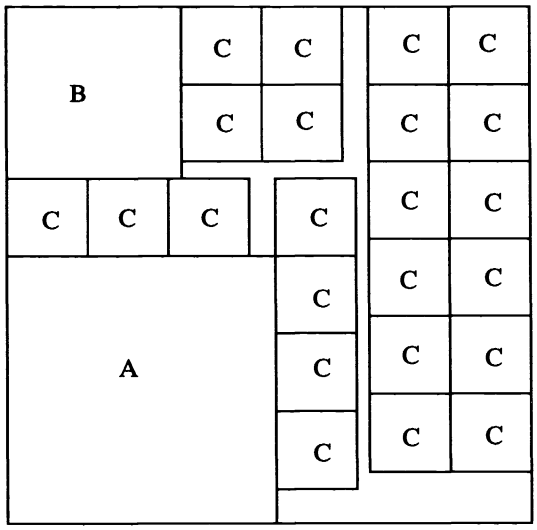
$$b_i = \begin{cases} \left(\frac{1}{3}, \frac{1}{3}, 1-(i-1)\xi\right) & \text{if } i \bmod 14 = 1, \\ \left(\frac{1}{6}-\delta, \frac{1}{6}-\delta, 1-(i-1)\xi\right) & \text{if } i \bmod 14 = 2, 3, \cdots, 13, \\ \left(\frac{1}{k}, \frac{1}{k}, 1-(i-1)\xi\right) & \text{if } i \bmod 14 = 0. \end{cases}$$

By properly choosing k and δ, we can enforce $L = (b_1, b_2, \cdots, b_{378n})$ to be partitioned into $27n$ sublists each having 14 boxes (cf. Lemma 2). Therefore,

$$C_1(I) = 43n - 189n(27n-1)\xi.$$

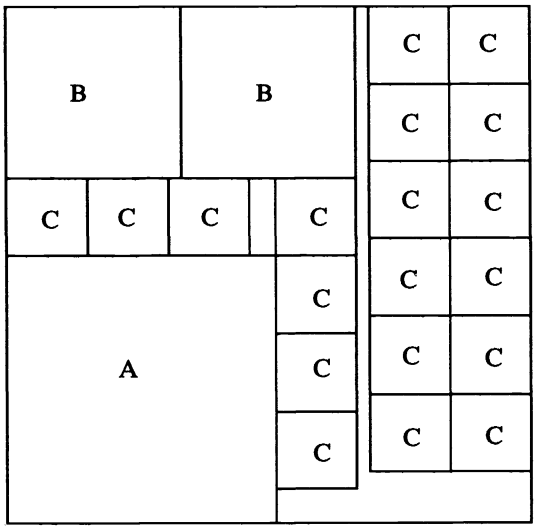
On the other hand, there are $16n+1$ levels in the optimal packing where $5n$ levels are of the same type whose bird's-eye view is illustrated in Fig. 6(a), $11n$ levels are of another type whose bird's-eye view is illustrated in Fig. 6(b), and one level for boxes with bottom $(1/k, 1/k)$. □

The idea of algorithm C can also be used to handle boxes with small bottoms. Define algorithm C_m $(m \geqq 2)$ as follows. Given an instance $I \in \mathfrak{I}_m$, C_m divides I into



A: $(\frac{1}{2}+\delta, \frac{1}{2}+\delta)$
B: $(\frac{1}{3}, \frac{1}{3})$
C: $(\frac{1}{6}-\delta, \frac{1}{6}-\delta)$

(a) 5n levels of this type



(b) 11n levels of this type

FIG. 6. *Illustration of a possible packing* (*Theorem* 8).

two subsets: S_1 in which $1/(m+1) < x_i = y_i \leqq 1/m$ and S_2 in which $x_i = y_i \leqq 1/(m+1)$. Then C_m produces the packings of S_1 and S_2 by using NFDH_m^x and G_{m+1}, respectively. The two packings are finally combined to form a packing of I. We also define algorithm C_m^* $(m \geqq 2)$ as follows. Given an instance $I \in \mathfrak{S}_m^*$, C_m^* divides I into three subsets: S_1 in which $1/(m+1) < x_i \leqq 1/m$ and $y_i \leqq x_i$; S_2 in which $1/(m+1) < y_i \leqq 1/m$ and $y_i > x_i$, and S_3 in which $0 < x_i, y_i \leqq 1/(m+1)$. Then C_m^* produces the packings of S_1, S_2, and

S_3 by using NFDH_m^x, NFDH_m^y and G_{m+1}^*, respectively. Finally, the three packings are combined.

THEOREM 9. *For any $I \in \Im_m$ $(m \geqq 2)$ where no box has height exceeding Z, we have*

$$C_m(I) < \alpha'_m \cdot \text{OPT}(I) + 2Z$$

where $\alpha'_m = ((m+1)/m)^2$. Furthermore, the bound α'_m is tight.

Proof. Note that in the packing of S_1, each level except the last one contains exactly m^2 boxes. Since the bottom area of each box is larger than $(1/(m+1))^2$, the utilization of each level except the last one is larger than $(m/(m+1))^2$. By Lemma 2, this is also true in the packing of S_2. Thus it is easy to verify the inequality in the theorem. To show the tightness of α'_m, we can use the instance I_{m+1} constructed in the proof of Theorem 5. □

THEOREM 10. *For any $I \in \Im_m^*$ $(m \geqq 2)$ where no box has height exceeding Z, we have*

$$C_m^*(I) < \beta'_m \cdot \text{OPT}(I) + 3Z$$

where $\beta'_m = (m+1)/(m-1)$ which is a tight bound.

Proof. By Lemmas 3 and 4, in the packing of S_i $(i = 1, 2, 3)$, the utilization of each level except the last one is larger than $(m-1)/(m+1)$. The instance I_{m+1}^* in the proof of Theorem 6 can be used to show that β'_m is tight. □

Note that $\alpha'_m = \alpha_{m+1} < \alpha_m$, $m \geqq 2$ and $\beta'_m < \beta_m$, $m \geqq 3$, i.e., the performance of C_m $(m \geqq 2)$ is better than G_m and the performance of C_m^* $(m \geqq 3)$ is better than G_m^*. Table 1 summarizes the asymptotic performance bounds for all algorithms discussed in the paper. Note that we let $G_1^* = G$ and $C_1^* = C$, and there is no Algorithm G_2^*.

5. Summary. We have studied the three-dimensional packing problem in this paper. The problem is more complex than the one- and two-dimensional packing problems. We demonstrate its difficulty by showing that Algorithms NFDH and FFDH do not work because they do not have enough heuristics. Algorithm G is developed based on the analysis of the two-dimensional packing strategy L^x (L^y) in [15]. It is then refined to Algorithm C. Both algorithms are adapted for the cases when all boxes have square bottoms (G_1, C_1) and when all boxes have small bottoms $(G_m, C_m, C_m^*$ for $m \geqq 2$ and G_m^* for $m \geqq 3)$. Algorithms in the C family are based on the good performance of NFDH^x, NFDH^y, and their variants (NFDH_m^x, NFDH_m^y for $m \geqq 2$) in certain cases (e.g., for instances in $\Im_m^x$ and $\Im_m^y$).

All algorithms discussed in this paper are based on the level-by-level layer-by-layer packing scheme (LL^x, LL^y). The heuristic in the z-dimension is basically nonincreasing

TABLE 1
Summary of results.

m	G_m^*	C_m^*	G_m	C_m
1	4.571[1]	3.25	4	2.6875
2		3	4	2.25
3	3	2	2.25	1.778
4	2	1.667	1.778	1.5625
5	1.667	1.5	1.5625	1.44
⋮	⋮	⋮	⋮	⋮
m	$\dfrac{m}{m-2}$	$\dfrac{m+1}{m-1}$	$\left(\dfrac{m}{m-1}\right)^2$	$\left(\dfrac{m+1}{m}\right)^2$

[1] Tightness is unknown.

height. However, sorting is done within each subset of I instead of the whole instance I. Algorithms differ from each other in the way they divide I into subsets. Thus the main attention is focused on how to increase the utilization in all levels except for a constant number of levels.

Appendix A. Because of difficulties in drawing figures to illustrate three-dimensional packings, and also to avoid confusion, we define some terms used throughout this paper in a mathematical way. By using a three-dimensional coordinate system, the box B can be regarded as a region

$$B=[0, w]\times[0, l]\times[0, \infty).$$

A *packing* $\wp$ of a set of boxes $I=\{b_1, b_2, \cdots, b_n\}$ into B is a mapping

$$\wp: I\mapsto B$$

such that

$$p^x(b_i)+x_i\leqq w \quad \text{and} \quad p^y(b_i)+y_i\leqq l$$

where

$$\wp(b_i)=(p^x(b_i), p^y(b_i), p^z(b_i)), \qquad 1\leqq i\leqq n,$$

i.e., each box is entirely enclosed in the box B and the packing is orthogonal and oriented. In addition, if $R(b_i)$ is defined as

$$R(b_i)=[p^x(b_i), p^x(b_i)+x_i)\times[p^y(b_i), p^y(b_i)+y_i)\times[p^z(b_i), p^z(b_i)+z_i),$$

then

$$\forall i, j, \quad 1\leqq i\neq j\leqq n: R(b_i)\cap R(b_j)=\varnothing,$$

i.e., no two boxes in L can overlap in a packing $\wp$. The *height* of a packing $\wp$ is

$$H(\wp)=\max_{1\leqq i\leqq n}(p^z(b_i)+z_i).$$

Hence the 3D *packing problem* can be formally defined as follows: "Given w, l, and a set of boxes $I=\{b_1, b_2, \cdots, b_n\}$, find a packing $\wp$ of I into B such that $H(\wp)$ is minimized."

All algorithms developed in this paper divide a set of boxes I into two or more disjoint subsets $I_1, I_2, \cdots, I_v$, then generate their packings $\wp_1, \wp_2, \cdots, \wp_v$ separately and finally combine these packings to give a packing of I. Suppose

$$\wp_i: I_i\mapsto B, \qquad 1\leqq i\leqq v,$$

where

$$\wp_i(b)=(p_i^x(b), p_i^y(b), p_i^z(b)), \qquad b\in I_i.$$

Define the *concatenation* (or *combination*) of the v packings $\wp_1, \wp_2, \cdots, \wp_v$, denoted as $\wp_1\|\wp_2\|\cdots\|\wp_v$, be a packing $\wp$ such that

$$\wp: I\mapsto B$$

where

$$\wp(b)=\left(p_i^x(b), p_i^y(b), \sum_{j=1}^{i-1}H(\wp_j)+p_i^z(b)\right), \qquad b\in I_i, \quad 1\leqq i\leqq v.$$

Obviously,

$$H(\wp)=H(\wp_1\|\wp_2\|\cdots\|\wp_v)=H(\wp_1)+H(\wp_2)+\cdots+H(\wp_v).$$

Appendix B. All algorithms discussed in this paper adopt a level-by-level layer-by-layer (LL) packing scheme. An LL packing consists of a set of levels; each level consists of a set of layers and each layer consists of a set of boxes. A *level* in a packing $\wp$ is a region

$$B = [0, w] \times [0, l] \times [Z_1, Z_2)$$

in which there is a set S of boxes such that

$$\forall b \in S: p^z(b) = Z_1 \text{ and } Z_2 - Z_1 = \max_{b \in S} (z).$$

A *layer* in a level is a region

$$B = [0, w] \times [Y_1, Y_2) \times [Z_1, Z_2)$$

in which there is a set S of boxes such that

$$\forall b \in S: p^y(b) = Y_1 \text{ and } p^z(b) = Z_1 \text{ and } Y_2 - Y_1 = \max_{b \in S} (y) \text{ and } Z_2 - Z_1 = \max_{b \in S} (z).$$

Finally, a formal description of the LL^x packing algorithm is given in Algorithm 4 where (x, y, z) is the position for the next box, $y_{\max}$ is the maximum depth of all boxes in the current layer and $z_{\max}$ is the maximum height of all boxes in the current level. Note that Algorithm LL^y can be defined accordingly if layers are parallel to the y-dimension.

Procedure LL^x.
    $(x, y, z, y_{\max}, z_{\max}) \leftarrow (0, 0, 0, 0, 0)$
    for $i \rightarrow 1$ **to** n **do**
        if $(x + x_i > w)$ or $(y + y_i > l)$ **then**
            if $y + y_{\max} + y_i \leqq l$ **then**
                $y \leftarrow y + y_{\max}$
                $(x, y_{\max}) \leftarrow (0, 0)$
            else
                $z \leftarrow z + z_{\max}$
                $(x, y, y_{\max}, z_{\max}) \leftarrow (0, 0, 0, 0)$
            end if
        end if
        $\wp(b_i) \leftarrow (x, y, z)$
        $(x, y_{\max}, z_{\max}) \leftarrow (x + x_i, \max(y_{\max}, y_i), \max(z_{\max}, z_i))$
    end for
end LL^x

ALGORITHM 4

REFERENCES

[1] B. S. BAKER, E. G. COFFMAN, JR., AND R. L. RIVEST, *Orthogonal packings in two dimensions*, SIAM J. Comput., 9 (1980), pp. 846-855.

[2] B. S. BAKER, D. J. BROWN, AND H. P. KATSEFF, *A 5/4 algorithm for two-dimensional packing*, J. Algorithms, 2 (1981), pp. 348-368.

[3] B. S. BAKER AND J. S. SCHWARZ, *Shelf algorithms for two-dimensional packing problems*, SIAM J. Comput., 12 (1983), pp. 508-525.

[4] F. R. K. CHUNG, M. R. GAREY, AND D. S. JOHNSON, *On packing two-dimensional bins*, SIAM J. Algebraic Discrete Methods, 3 (1982), pp. 66-76.

[5] E. G. COFFMAN, JR., M. R. GAREY, D. S. JOHNSON, AND R. E. TARJAN, *Performance bounds for level-oriented two-dimensional packing algorithms*, SIAM J. Comput., 9 (1980), pp. 808–826.

[6] E. G. COFFMAN, JR., M. R. GAREY, AND D. S. JOHNSON, *Approximation algorithms for bin-packing—an updated survey*, *in* Algorithm Design for Computer System Design, G. Ausiello, M. Lucertini, and P. Serafini, eds., Springer-Verlag, New York, 1984, pp. 49–106.

[7] M. R. GAREY, R. L. GRAHAM, D. S. JOHNSON, AND A. C. YAO, *Resource constrained scheduling as generalized bin packing*, J. Combin. Theory. Ser. A, 21 (1976), pp. 257–298.

[8] M. R. GAREY AND D. S. JOHNSON, *Approximation algorithms for combinatorial problems: an annotated bibliography*, in Algorithms and Complexity: New Directions and Recent Results, J. F. Traub, ed., Academic Press, New York, 1976, pp. 41–52.

[9] R. L. GRAHAM, *Bounds on multiprocessing timing anomalies*, SIAM J. Appl. Math., 17 (1969), pp. 263–269.

[10] D. S. JOHNSON, A. DEMARS, J. D. ULLMAN, M. R. GAREY, AND R. L. GRAHAM, *Worst-case performance bounds for simple one-dimensional packing algorithms*, SIAM J. Comput., 3 (1974), pp. 299–325.

[11] R. M. KARP, M. LUBY, AND A. MARCHETTI-SPACCAMELA, *A probabilisitc analysis of multidimensional bin packing problems*, in Proc. 16th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1984, pp. 289–298.

[12] K. LI AND K. H. CHENG, *Complexity of resource allocation and job scheduling problems in partitionable mesh connected systems*, in Proc. 1st IEEE Symposium on Parallel and Distributed Processing, IEEE Computer Society, Dallas, TX, May 1989, pp. 358–365.

[13] ———, *Job scheduling in partitionable mesh connected systems*, in Proc. 18th International Conference on Parallel Processing, Vol. 2, August 1989, pp. 65–72.

[14] ———, *Job scheduling in* PMCS *using a* 2DBS *as the system partitioning scheme*, in Proc. 19th International Conference on Parallel Processing, August 1990, to appear.

[15] A. MEIR AND L. MOSER, *On packing of squares and cubes*, J. Combin. Theory, 5 (1968), pp. 126–134.

[16] S. SAHNI, *Algorithms for scheduling independent tasks*, J. Assoc. Comput. Mach., 23 (1976), pp. 116–127.

[17] A. C. YAO, *New algorithms for bin packing*, J. Assoc. Comput. Mach., 27 (1980), pp. 207–227.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 868-882, October 1990

007

POSTORDER DISJOINT SET UNION IS LINEAR*

JOAN M. LUCAS†

Abstract. Any instance of the disjoint set union problem of size n, using any union strategy, that does one find per node in postorder has total cost $O(n)$. This special case, when the finds are restricted to occur in postorder, is related to the behavior of such self-adjusting data structures as the splay tree and the pairing heap.

Key words. disjoint set union, union-find, data structures, splay tree

AMS(MOS) subject classifications. 68, 05

1. Introduction. In this exposition we consider the special case of the disjoint set union problem when the find operations are performed in a "unidirectional" manner. In such an instance, the find operations are performed left to right. For any node x, all finds from descendants of the leftmost child of x precede all finds from descendants of the second leftmost child of x, etc. Equivalently, the finds are performed in postorder.

Hart and Sharir first considered the complexity of find operations which are constrained to be performed in postorder. They studied the relationship between the length of Davenport-Schinzel sequences and the cost of performing a generalized form of path compressions in postorder. These sequences are related to the analysis of linear differential equations and the complexity of many problems that arise in computational geometry [1]. The special case of postorder finds is also connected to several interesting open questions in the analysis of data structures. Postorder path halving is an important component of the behavior of both splay trees [2], [3] and pairing heaps [4].

The disjoint set union problem consists of a universe of n elements, partitioned into sets. Without loss of generality we can consider the initial sets to be singletons. The only permitted operations are:

Union (A, B): combine the two sets A and B into one set.

Find (x): return the name of the set containing element x.

In the most popular data structure used to implement these operations each element is represented by a node, and each set is represented by a rooted tree whose nodes are the elements in that set. See Fig. 1. Each node has pointer to its parent in the tree. The root of the tree serves as the name of the corresponding set. A union operation is performed by taking the roots of two trees and making one root a child of the other. The cost of this operation is $O(1)$.

A Find (x) operation is performed by following parent pointers up from x to the root of the tree. The path traversed by this operation is referred to as the find path. A find operation is usually defined to include some extra work. The parent pointers of nodes on the find path are changed with the goal of speeding up future find operations. The following are two standard implementations of a find operation. Let $x_0, x_1, \cdots, x_k$ be the nodes on the find path, i.e., x_0 is the parameter of the find and x_k is the root.

(1) Path compression. For each node x_i, $0 \leqq i < k$, set the parent of x_i to x_k, i.e., all the nonroot nodes on the find path become children of the root.

(2) Path halving. For each node x_i, $1 \leqq i < k$ and i odd, reset the parent of x_i to be x_{i+2} if this grandparent of x_i exists, i.e., if $i+2 \leqq k$.

* Received by the editors October 3, 1988; accepted for publication (in revised form) January 12, 1990.
† Department of Computer Science, Hill Center, Rutgers University, New Brunswick, New Jersey 08903.

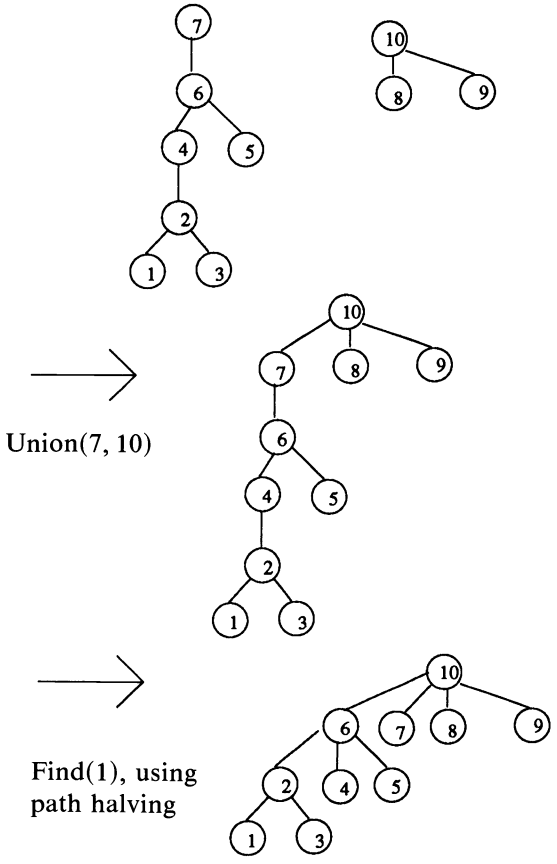


FIG. 1

The cost of a find is the number of nodes on the find path whose parent has changed. Note that the cost of a find, as defined here, differs from the standard definition, which is either the number of edges or the number of nodes on the find path, by only a constant factor.

In previous studies of variations of this problem [5]–[7], the trees representing the sets are unordered trees; for these problems the left-to-right order in which the children of a node appear is irrelevant. For our purposes this ordering is important. We shall use the same data structure described above but we make the additional assumption that the trees under consideration are ordered trees.

We shall use the following notation. For any node x in some tree T, we define the subtree rooted at x, denoted $T(x)$, to be the tree induced by deleting nodes that are not descendants of x. We consider x to be an ancestor and descendant of itself. Let T be an ordered tree with root r. Let the children of R be $x_1, x_2, \cdots, x_l$, $l \geqq 0$, where x_i appears to the left of x_j if $i < j$. Then we define the postorder sequence of T, denoted POST (T), to be POST $(T(x_1))$ POST $(T(x_2)) \cdots$ POST $(T(x_l))r$. The postorder sequence of an ordered forest of trees $T_1, T_2, \cdots, T_l$, where T_i appears to the left of T_j if $i < j$, is simply POST (T_1) POST $(T_2) \cdots$ POST (T_l).

We shall assume that initially the input consists of an ordered forest of n single node trees. For each node we associate with it its position in the postorder sequence of the initial forest. We do not distinguish between a node and its postorder position.

We shall assume from this point on that any forest or tree under discussion is ordered.

We are interested in performing the disjoint set union operations so as to preserve the postorder sequence of the forest.

Thus the union operation is now defined to take two tree roots r_1 and r_2, $r_1 < r_2$, and make r_1 the leftmost child of r_2. Any tree of size n can be constructed by performing $n-1$ of these ordered unions.

The Find (x) operation in tree T is defined as before, i.e., the same parent pointer updates are performed, but with the added restriction that the resulting tree is ordered so that it has the same postorder sequence as T. Note that it is possible to implement a path compression or halving operation in such a way as to preserve the postorder sequence of the tree if and only if the following holds. For every node x_j, $0 < j < k$, on the find path, if some x_i, $i < j$, has its parent reset to some x_h, $h > j$, then x_{j-1} is the leftmost child of x_j. In particular, if x_0 is the least node in the tree in postorder, then this find can be implemented so as to preserve the postorder.

Path halving has the advantage that it can be implemented using only one pass over the find path, and not two as are needed for path compression. Another variant of path compression that requires only one pass over the find path is path splitting. In this case every node on the find path has its parent reset to its grandparent. We do not consider this variant since it cannot be implemented so as to preserve the postorder sequence of the tree, and indeed a sequence of n finds done in postorder with respect to the initial tree can cost $\Omega(n \log n)$ [5].

We can define a very general reshaping of the find path as follows.

(3) Path shrinking. The parent pointer of any node x_i, $0 \leqq i < k$, can be reset to any ancestor of x_i so long as the resulting tree has the same postorder sequence as the inital tree. The cost of this operation is the number of nodes on the find path whose parent is changed.

Note that path compression and path halving are special cases of path shrinking.

In § 2 we show the main result: that the cost of postorder disjoint set union, using standard find strategies, is linear. Section 3 discusses the results of some experimental work regarding path compressions done in postorder that do not correspond to a union-find problem. Section 4 considers the more general path shrinking strategies. Section 5 discusses the relationship between these problems and open questions concerning splay trees and pairing heaps.

2. Linearity of postorder path compression and halving. To simplify the analysis of the cost of any sequence of union and find operations, we usually consider all the unions as having been performed first. Then to simulate the find operations the definition of the operation must be altered to that of a *partial find.* Precisely the same parent pointer updates are performed for each operation but now the find path of Find (x_0) need not be the path from x_0 to the root of the tree containing x_0, but it is the path from x_0 to some ancestor of x_0, which is referred to as the root of the find. All other nodes on the find path are referred to as nonroot nodes of the find.

The following lemma describes when a sequence of partial finds corresponds to a sequence of intermixed union and find operations.

LEMMA 1 (Rising Roots Condition). *A sequence of partial finds corresponds to a sequence of intermixed union and find operations if and only if, for any node x, if x appears as a nonroot node on the find path of* Find (u), *then for any future find from y,*

$y > u$. If x is an ancestor of y at that time, then x is a nonroot node on the find path of FIND (y).

Proof. The necessity of this condition is trivial. To prove its sufficiency, consider a sequence Σ of partial finds performed on a tree whose edges can be either solid or dashed. Suppose the sequence conforms to the following conditions:

(1) The dashed edges form a connected subtree containing the root of the tree.

(2) If the dashed edge from x to its parent is converted to a solid edge, then x is a leaf in the dashed edge subtree.

(3) Every partial find contains only solid edges on the find path and the root of the find is either the root of the tree or has a dashed edge to its parent.

Then clearly Σ corresponds to a sequence of intermixed union and find operations. A node in this instance corresponds to a root in the union-find problem if it is the root of the tree or has a dashed edge to its parent. The conversion of the edge (x, y) to a solid edge corresponds to a union between the root nodes x and y.

Thus to prove the lemma it is sufficient to show that, given any sequence of partial finds that obeys the condition of the lemma, the edges of the trees created by the sequence can be divided into solid and dashed edges so that the three conditions above hold.

The proof is by induction on the number of finds. The induction hypothesis will be that all three conditions hold before the ith find and also that:

(4) An edge (x, parent (x)) is solid if and only if during some previous find x, or an ancestor of x at that time, appeared as a nonroot node on that find path.

Note that once x has an ancestor that has appeared as a nonroot node on a find path, then at all future times x has such an ancestor, though not necessarily the same node.

Initially we let all the edges be dashed. Clearly the hypothesis holds before the first find. Consider the ith find. Let the find path be $x_0, x_1, \cdots, x_k$. Then there is some node x_j, $0 \leqq j \leqq k$, such that all the edges on the find path below x_j are solid and all those above are dashed. The edge from x_k to its parent (if it exists) cannot be solid, since in that case x_k, or one of its current ancestors, has previously appeared as a nonroot node, contradicting the claim that the sequence obeys the condition of the lemma.

Before performing the find we convert to solid edges all the dashed edges in the subtree rooted at x_{k-1} and the edge (x_{k-1}, x_k). Clearly the dashed edges in the resulting tree form a connected subtree containing the root and these conversions can be ordered so that when the edge from x to its parent is converted, x is a leaf node in the dashed edge subtree. Once these conversions are completed then the partial find is performed. All the edges on this find path are solid, and the root node of the find is either the root of the tree, or has a dashed edge to its parent. Also it is clear that condition (4) holds after the conversions and the partial find. □

We now assume that all the union operations are performed first, and without loss of generality we may assume that unions create a single tree T_0. Let Σ be a sequence of partial finds performed starting with T_0. Let T_i denote the tree created by performing the first i operations of Σ and then deleting all irrelevant nodes. A node x is irrelevant if it cannot lie on any future find path, i.e., if no descendant of x is the parameter of any future find.

For convenience we add to T_i another node, with label $n+1$, that is the parent of the node labeled n, i.e., the real root of the tree. This dummy root is never the root of a find path. Without loss of generality we can assume that at least one find path

has root n, since the root of the final find path can be changed to n, and this change can only increase the cost of Σ.

The following is our main result.

THEOREM 1. *Let T_0 be any ordered tree containing n nodes labeled 1 to n in postorder. Let Σ be any sequence of n partial finds that satisfies the rising roots condition, does one find for each node in postorder, and uses the path compression (respectively, path halving) strategy. Then the total cost of Σ is $O(n)$.*

Note that we do not require the initial tree to be the result of weighted unions. The result holds for *any* initial tree.

To prove the theorem we show how to divide the tree into disjoint subproblems. The cost of the original problem instance $P=(T_0, \Sigma)$ will equal the cost of the subproblems plus some linear term. The subproblems induced will be of the same form as the original problem with the added restriction that the roots of the partial finds are fixed to be the root of the tree. Then to complete the theorem it will be sufficient to establish the linear bound for the more restricted version.

LEMMA 2 (Path Compression with Fixed Root). *Let T_0 be any ordered tree containing n nodes labeled 1 to n in postorder. Let Σ be any sequence of n finds, using path compression, such that the root of each find is the root of the tree. Then the total cost of Σ is $O(n)$.*

Proof. Each find of cost k increases the number of children of the root by k. Thus the total cost is at most $n-1$. □

Note that this lemma does not require that the finds be done in postorder.

LEMMA 3 [8] (Path Halving with Fixed Root). *Let T_0 be any ordered tree containing n nodes labeled 1 to n in postorder. Let Σ be any sequence of n finds, using path halving, that does one find for each node in postorder such that the root of each find is the root of the tree. Then the total cost of Σ is $O(n)$.*

Proof. This problem is identical to the sequential access problem in splay trees. In that problem the nodes of a binary tree are labeled from 1 to n in symmetric order. A splay operation is then performed on each of the nodes in that order, after which that node is deleted from the tree. This is precisely equivalent to an instance of postorder path halving with a fixed root. The corresponding tree in the path halving problem is derived from the binary tree by the natural correspondence, i.e., by setting the parent of each node to be its nearest ancestor in the binary tree with a greater label. See Figs. 2 and 3. Since the parameter node of the find (node 1 in Figs. 2 and 3) never again appears on any find path, we consider it as being deleted from the tree. Then by the Sequential Access Lemma of [8], the cost of such a sequence is $O(n)$. □

Now to prove Theorem 1 we show how to define the subproblems.

First we note that if the root of the tree, node n, has more than one child, then the subproblem consisting of n, some child x of n, and the subtree rooted at x is disjoint from all the other operations. Thus without loss of generality we assume that the root has only one child.

Let x be a child of y in T_0. If every partial find from a node that is a descendant of x in T_0 has a descendant of y as the root of its find path, then the problem can be divided into two subproblems. The first subproblem has as its initial tree the subtree of T_0 induced by x, y, and the descendants of x, and includes the partial finds of x and its descendants. The second subproblem has as its initial tree T_0 with the subtree rooted at x deleted, and includes the finds of all of these nodes. These problems are disjoint with respect to the edges.

Thus to prove the theorem it is sufficient to show that the result holds when Σ satisfies the stronger condition that for every edge (x, y), $y \neq n$, in T_0, at least one

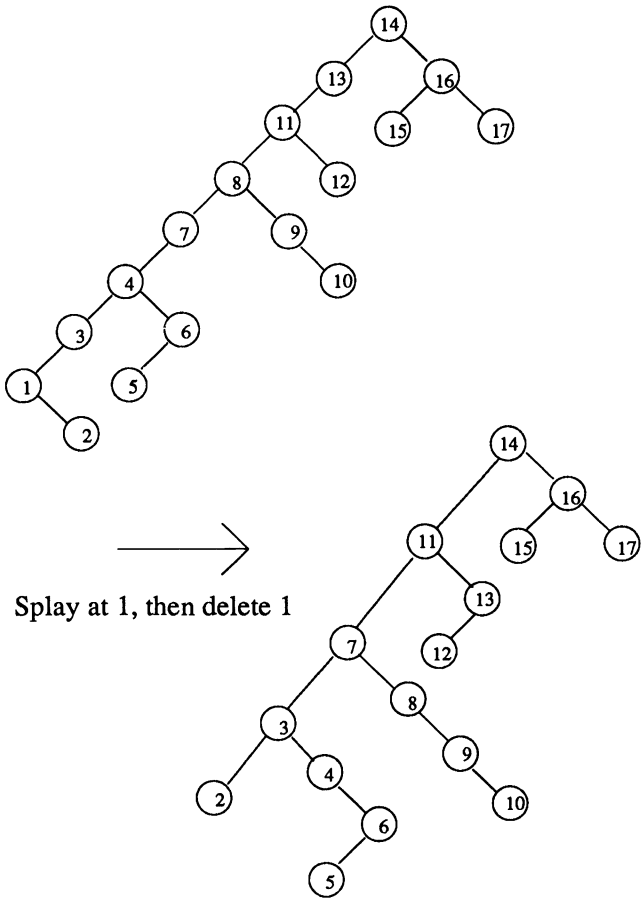


FIG. 2

partial find from a descendant of x in T_0 contains on its find path a node that is a proper ancestor of y in T_0. Naturally this condition cannot hold for the one edge from the root to its child. Without loss of generality we assume now that this interdependency condition holds.

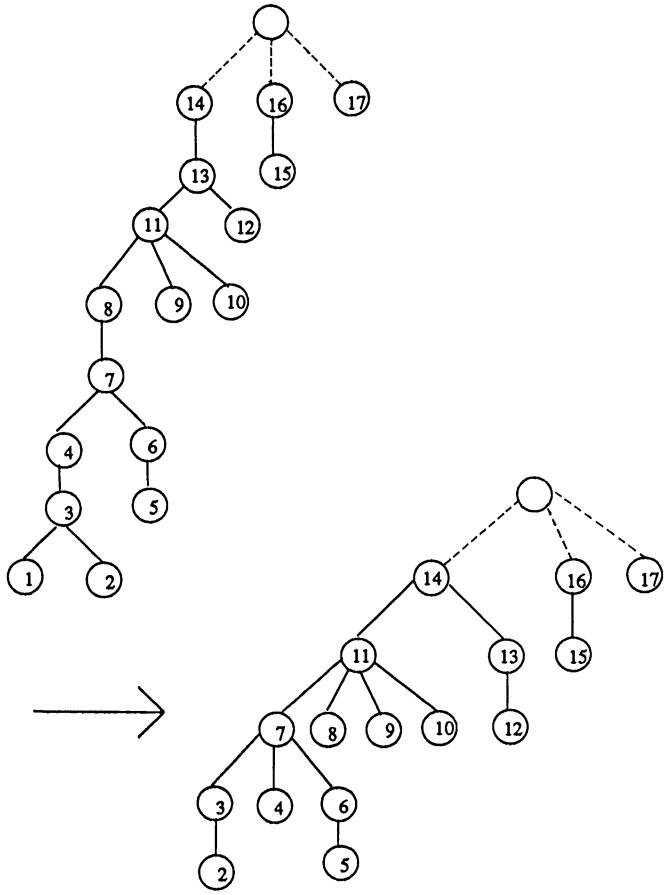
In the proof of Theorem 1 we shall make use of the following lemmas.

LEMMA 4. *For any node u in T_i, if some ancestor of u in T_i has appeared as a nonroot node on the find path of* FIND (h), *$h \leqq i$, then in every tree T_j, $j \geqq i$, u has some ancestor in T_j that has appeared as a nonroot node on the find path of* Find (l), *for some $l \leqq j$.*

Proof. The proof is by induction. We show that if u has some ancestor v in T_i such that v has appeared as a nonroot node on the find path of some previous Find (h), $h \leqq i$, then u has such an ancestor in T_{i+1}.

Suppose the find path of Find $(i+1)$ contains no proper descendant of v. Then v is still an ancestor of u after Find $(i+1)$ is performed, i.e., in T_{i+1}, so the lemma holds.

Otherwise $i+1$ is a descendant of v in T_i. Let $i+1 = x_0, x_1, \cdots, x_k$, be the nodes on the find path of Find $(i+1)$. By the rising roots condition, v must apear on this find path as a nonroot node. Let x_j be the nearest ancestor of u that lies on the find path. Then x_j is a descendant of v, and so x_j must be a nonroot node of the find. But x_j is an ancestor of x in T_{i+1} so the lemma is proved. □



Find(1), then delete 1

FIG. 3

LEMMA 5. *For any node u in T_i, if some ancestor v of u in T_i has appeared as a nonroot node on the find path of* Find (h), $h \leqq i$, *the root of the find path of* Find (u) *is greater than v.*

Proof. Clearly if v is an ancestor of u when Find (u) is performed, then the find path must include v as a nonroot node. The root of Find (u) must be a proper ancestor of v, and thus, by the postorder numbering, be greater than v.

Otherwise there exist nodes w and y such that w and y are ancestors of u in T_i, $u \leqq w < v < y$, and some find operation Find (l), $i < l < u$, updates the parent pointer of w to be y. Choose the maximum such l, and given that value of l, choose the minimum such w. Then w and y are ancestors of u in T_{u-1}, and w has appeared as a nonroot node on a previous find path (Find (l)), therefore the find path of Find (u) must contain $y > v$, and the lemma is proved. □

To prove Theorem 1 we divide the original problem $P = (T_0, \Sigma)$ into subproblems $P_1 = (\tau_1, \Sigma_1)$, $P_2 = (\tau_2, \Sigma_2), \cdots, P_k = (\tau_k, \Sigma_k)$. We shall distinguish certain nodes $1 = z_0 < z_1 < z_2 < \cdots < z_k = n+1$, the choice of which will depend on the nature of the partial finds of Σ. The subproblem P_i will contain the nodes x such that $z_{i-1} \leqq x \leqq z_i$ and will perform one find for each node. The edges of the original tree are partitioned

by the subproblems, though the nodes are not. Each subproblem P_i is an instance of finds being performed in postorder where the root of each find must be the root of the tree.

The initial tree for subproblem P_i, denoted τ_i, is formed as follows. Take the tree formed by performing the first $z_{i-1}-1$ partial finds of Σ on T_0 (i.e., $T_{z_{i-1}-1}$), delete all nodes that are not descendants of z_i, and finally contract the edge from z_i to its leftmost child, denoted y_i, and denote the resulting node r_i. This node is the root of τ_i. See Fig. 4. In this figure a dotted line from a node y down to a node x denotes that $x \leqq y$ and that there is a path from y to x through current leftmost child edges. The number of nodes on this path, including the endpoints, is greater than or equal to one. A node x with an attached triangle denotes the subtree induced by all descendants of x that are not descendants of the leftmost child of x.

The finds for each subproblem are defined by slightly modifying the partial finds of Σ. Let x_0, $z_{i-1} \leqq x_0 < z_i$, be a node of the subproblem P_i. The partial find from x_0 in P_i is defined as follows. Consider the find path $x_0, x_1, \cdots, x_h$ of Find (x_0) in the original problem. Recall that a partial find operation consists of repeatedly performing

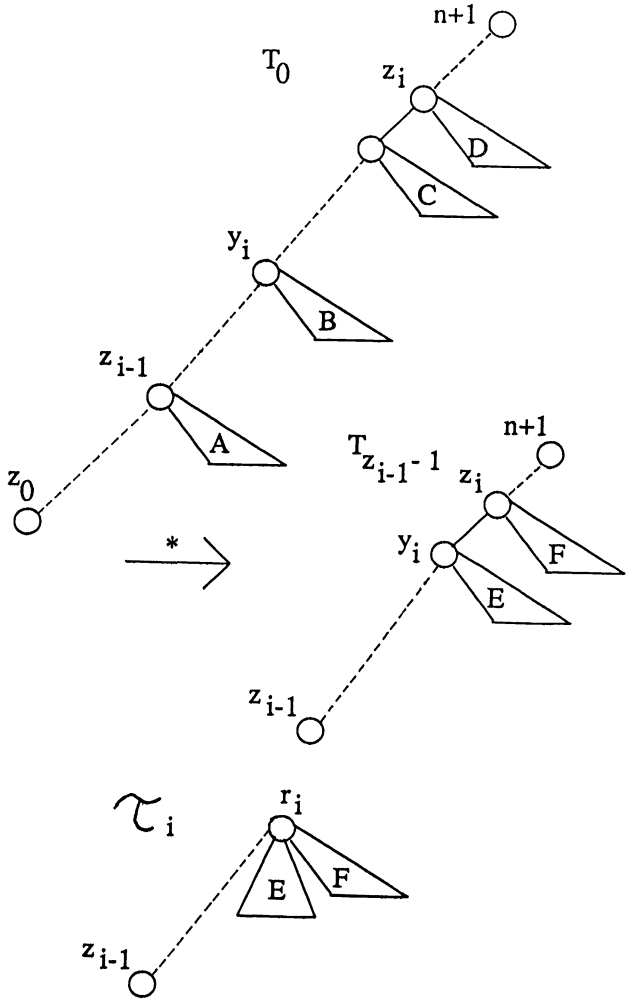


FIG. 4

the following edge update operation: reassign a node's parent pointer to some proper ancestor of its parent, at a cost of 1 per update. Suppose the parent of x_j is reset to x_l, $0 \leqq j < l-1 < h$.

If both x_j and x_l are at most z_i, then the parent of x_j is updated as part of Find (x_0) in the subproblem P_i. The parent of x_j is updated to x_l if $x_l \neq y_i$ and $x_l \neq z_i$, otherwise it is updated to r_i. If $x_j < z_i < x_l$, then as part of Find (x_0) in P_i the parent of x_j is reset to be r_i. These updates are called subproblem updates. Note that possibly $x_{j+1} = y_i$ or $x_{j+1} = z_i$, in which case the updating of x_j's parent could be a null operation in the subproblem. We call such an update an undercounted update. There can be at most two such undercounted updates per find.

Finally, if both x_j and x_l are at least z_i, then this update is not performed as part of any of the subproblems, and is called a nonsubproblem update.

Clearly the operation Find (x_0) in subproblem P_i is well defined and is a path compression (respectively, path halving) operation in the subproblem.

To complete the definitions of the subproblems P_i, we show how we identity the nodes z_i. We define z_0 to be the least node in the postorder of T_0, i.e., node 1, and z_i, $i > 0$, to be the parent of the root of the find path for Find (z_{i-1}). That is, y_i is the root of Find (z_{i-1}). Note that $z_{i-1} \neq z_i$, and thus none of the subproblems are empty and this procedure is guaranteed to produce a finite number of subproblems. Each find in Σ has a corresponding find operation in one of the subproblems.

Clearly, z_j is a proper ancestor of z_i in T_0 if $i < j$, since z_j is a proper ancestor of z_{j-1} in $T_{z_{j-1}-1}$ and the ancestors of a node can only decrease as a set due to the find operations.

We now observe that each subproblem P_i is an instance of postorder path compressions (respectively, path halvings) such that the root of each find path is the root of the tree.

Let x be some child of r_i in τ_i, i.e., the parent of x in $T_{z_{i-1}-1}$ is $p(x) = y_i$ or $p(x) = z_i$. Let u be any descendant of x in τ_i. A node can only lose descendants due to find operations, therefore u is a descendant of x in T_j for all $j < z_{i-1}$.

Suppose that x is not a child of $p(x)$ in T_0. Then x is a nonroot node on the find path of some Find (l), $l < z_{i-1}$. Then u is a descendant of x in T_l. so by Lemma 5, the root of Find(u) in Σ is greater than x, i.e., at least $p(x)$. Therefore the corresponding find in P_i roots at r_i.

Suppose that x is a child of $p(x)$ in T_0, but is not the leftmost child. By the interdependency condition there must be some node l that is not a descendant of x in T_0 such that the operation Find (l) in Σ includes $p(x)$ as a nonroot node on the find path. Then u has an ancestor $(p(x))$ in T_l that has appeared as a nonroot node on a previous find path, so by Lemma 5 Find (u) in Σ must root at some node greater than $p(x)$. Therefore the corresponding find in P_i roots at r_i.

Finally, suppose that x is the leftmost child of $p(x)$ in T_0. Let $l \leqq z_{i-1}$ be the minimum value such that the find path of Find (l) has root node $r > x$. Then x is a nonroot node on the find path of Find (l) and x is an ancestor of u in T_l, so by Lemma 5 Find (u) in Σ must root at some node greater than x. Therefore the corresponding find in P_i roots at r_i.

Thus each subproblem is an instance of postorder path compressions (respectively, halvings) with a fixed root, and thus has total cost linear in the size of the subproblem.

Clearly the cost due to undercounted updates is $O(n)$. It remains to be shown that the total cost of the nonsubproblem updates is $O(n)$.

We can account for these nonsubproblem updates using a simple amortized analysis. We assign credits to nodes in T_l, where $z_{i-1} \leqq l < z_i$ for some i, as follows:

each ancestor of z_i has one credit. We use these credits to pay for the nonsubproblem updates. The total cost of these updates is then bounded by the total number of credits in T_0, i.e., by n.

Consider the partial find operation performed on T_l. Each nonsubproblem update reduces the number of ancestors of z_i by at least one, thus the credits freed by these updates can be used to pay for these updates. If $l = z_i - 1$, then after Find (l) the number of credits changes from the number of ancestors of z_i in T_{z_i-1} to become the number of ancestors of z_{i+1} in T_{z_i-1}. But z_{i+1} is an ancestor of z_i in T_{z_i-1}, by the definition of the z nodes, therefore this crossing of the boundary between subproblems can only free up credits.

Thus the total number of nonsubproblem parent updates is bounded by the number of ancestors of z_1 in T_0, i.e., by n, and the theorem is proved.

It would be interesting to know whether the theorem holds when the rising roots condition is removed, i.e., when the root of a partial Find (x) can be *any* ancestor of x. Hart and Sharir posed this as an open problem in [1]. Such a conclusion holds for the standard problem definition of the disjoint set union problem. In this problem the postorder condition is dropped, but unions must be performed in a weighted manner. In this case an upper bound of $O(n\alpha(n))$ on the total cost, using path compression or path halving, has been proved [5] for *any* sequence of n partial finds, regardless of whether they correspond to a sequence of intermixed union and find operations.

Hart and Sharir studied the following more general problem. Let T_0 be any ordered tree of n nodes. Let Σ be a sequence of partial finds, one per node, done in postorder. The finds are implemented using the following special case of path shrinking. For every node x on the find path, if the parent of x is updated, then it must be updated to be the root of the find path.

Hart and Sharir proved a $\Theta(n\alpha(n))$ bound for the complexity of this problem. A similar, but more general, treatment appears in [9]. In the conclusion of their work, Hart and Sharir posed as an open problem determining the cost of a sequence of n path compressions performed in postorder. In this paper we have shown a linear bound on a special case of this open problem: the disjoint set union case.

In [10] and [11] Loebl and Nesetril have claimed a similar result. A theorem of that work states that the cost of an instance of the Postorder Set Union Problem is linear. No complete proof of this claim is given. Though the theorem is stated as a special case of the open problem of Hart and Sharir, the proof sketch provided does not make any reference to, nor any use of, the rising roots condition or any similar condition.

In [12] Loebl and Nesetril define another special case of the Hart and Sharir problem, the Postorder On-Line Set Union Problem, and provide a different argument to establish a linear bound for this problem. This case differs in that the finds are only constrained to appear in postorder with respect to the *current* tree, i.e., the union operations are not performed before all the find operations. This case is a generalization of the problem considered in this paper.

In the following section we explore the complexity of a sequence of postorder path compressions when the root of each partial find is unrestricted.

3. Experimental results. One observation we can make is that when analyzing the cost of sequences of unrestricted path compressions done in postorder we can assume that the initial tree consists of a single path with leaves attached to it.

LEMMA 6. *Let T_0 be any ordered tree and let Σ be any sequence of n partial finds using path compression done in the postorder. Then there exists a tree T_0' of size $\leqq 2n$*

such that every node in T_0' that is not a leaf lies on a single path, and there exists a sequence Σ' of $|T_0'|$ partial path compressions done in postorder starting from T_0' such that cost $(\Sigma) \leqq$ cost (Σ').

Proof. We show how to construct the tree T_0'. Assume that the nodes in T_0 are labeled 1 to n in postorder. All of these nodes will appear on the single path in T_0' that contains all the nonleaf nodes. This path is augmented by adding leaves to it on the left, i.e., all of the added leaves appear before the nodes 1 to n in the postorder of T_0'. The number of leaves attached to node $i < n$ equals the number of edges of the path in T_0 from node $i+1$ to the parent of i.

Σ' will first do partial finds from all the attached leaves. When these finds are completed the tree will be isomorphic to T_0. All remaining finds are identical to those in Σ. To duplicate the structure of T_0 each partial find from an attached leaf contains three edges on its find path. The effect of such a find from a leaf attached to node i, to the subtree of the nodes 1 to n, is to reset the parent of i to be its grandparent. See Fig. 5.

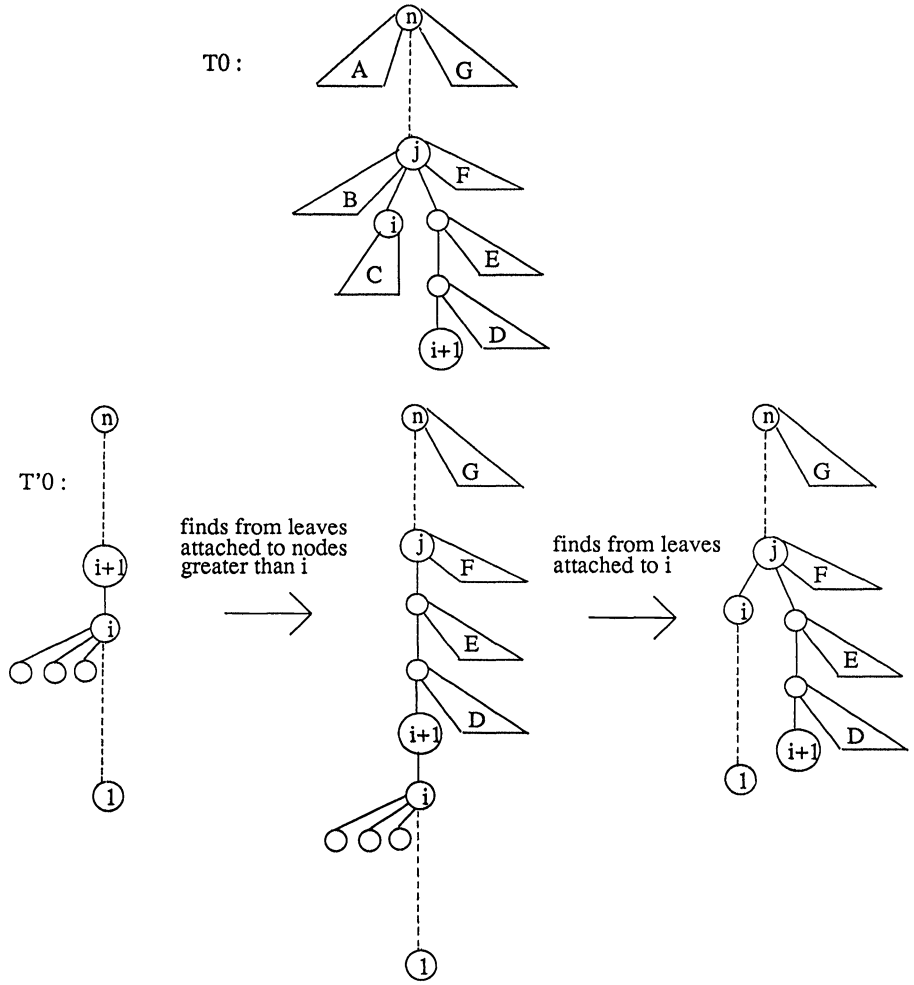


FIG. 5

There is a one-to-one correspondence between the attached leaves in T_0' and the edges in T_0, therefore $|T_0'| \leqq 2n$. Clearly the cost of Σ' is at least as great as Σ. □

In attempting to determine the bound on the cost of path compressions done in postorder, we performed the following experiment. Let the initial tree T_0 consist of a single path. From each of the nodes, in postorder, do a path compression of a fixed distance d, i.e., there are d edges on each find path.

Lemma 6 indicates that this is a reasonable choice for the initial tree. Also previous work indicates that doing compressions of a fixed distance can yield an expensive sequence. In [6] and [13] the disjoint set union problem, with weighted unions but without the postorder condition, was considered. Both derived nonlinear lower bounds doing one find per node, at a fixed cost each. Fischer [7] gave an $\Omega(n \log n)$ lower bound for path compression in an unbalanced tree. In this construction a find of length $\log n$ is done once for one half of the nodes. All of these proofs used sequences of finds that do not respect the postorder of the tree.

In our experiment we did a find from each node in postorder of the fixed compression length d, until we reached a node at depth $<d$, i.e., until it was not possible to continue. We calculated the fraction of nodes from which a compression was performed before this first failure occurred. After this first failure we continued to examine the nodes in postorder. For any node x, if the depth of x was greater than or equal to d then a compression of length d was performed. We also calculated the total fraction of nodes from which a compression was performed.

This experiment was performed for many different values of n. Surprisingly, these two fractions both appear to be constants, independent of n. Even more intriguing, these two fractions both appear to be very close to $1/(d-3)$.

A natural question to ask is whether these particular sequences of partial finds violate the rising roots condition, and indeed they do.

Table 1 details the results of our experiments. The initial tree consisted of a single path of 1,000,000 nodes. The first column gives the fixed compression distance d. The second column indicates the number of nodes, as a fraction of 1,000,000, from which finds were performed before the first failure. The third column gives the value of $1/(d-3)$. The fourth column indicates the fraction of nodes from which a compression was performed.

TABLE 1

d	Fraction before first failure	$1/(d-3)$	Fraction of successful compressions
4	0.99983600	1.00000000	0.99988400
5	0.51121500	0.50000000	0.55258376
6	0.33597000	0.33333333	0.34979110
7	0.25056100	0.25000000	0.25457578
8	0.20016000	0.20000000	0.20156561
9	0.16681500	0.16666667	0.16723051
10	0.14287700	0.14285714	0.14312043
11	0.12500600	0.12500000	0.12514738
12	0.11110700	0.11111111	0.11115433
13	0.09999700	0.10000000	0.10001830
14	0.09090500	0.09090909	0.09092227
15	0.08333000	0.08333333	0.08334025

In the following section we discuss why consideration of the generalized problem of postorder path shrinking is not useful towards obtaining a proof of linearity of arbitrary postorder path compression.

4. Postorder path shrinking. Path compression and path halving are special cases of the general path shrinking operation. Any upper bound for path shrinking applies to these operations as well.

THEOREM 2. *Let T_0 be any ordered tree containing n nodes labeled 1 to n in postorder. Let Σ be any sequence of n partial finds using path shrinking that does one find for each node in the postorder. Then the total cost of Σ is $O(n \log n)$.*

Proof. We divide the original problem into subproblems P_1 and P_2, each of which is an instance of postorder path shrinking. The first subproblem P_1 contains the nodes 1 through $\lfloor n/2 \rfloor$ and consists of the finds of all of these nodes. P_2 contains the nodes $\lceil n/2 \rceil + 1$ through n and consists of the finds of these nodes. The initial tree for P_2 is the tree that results from doing the first $\lceil n/2 \rceil$ finds.

As in the proof of Theorem 1, any edge update that assigns the parent of node $x \leqq \lfloor n/2 \rfloor$ a value greater than $\lfloor n/2 \rfloor$ is replaced by an edge update that sets the parent of x to be the root of the tree. This could cause an undercount of the true cost of the sequence by at most $\lfloor n/2 \rfloor - 1$.

We must also account for edge updates that do not belong to either subproblem, i.e., the update of the parent of $x > \lfloor n/2 \rfloor$ during the find of a node $y \leqq \lfloor n/2 \rfloor$. Each such update decreases the number of ancestors of the node $\lfloor n/2 \rfloor + 1$ by at least one. So there are at most $\lfloor n/2 \rfloor$ of these.

Therefore the total cost, $C(n)$, of any sequence of path shrinkings done in postorder is bounded by the recurrence relation $C(n) \leqq 2C(n/2) + n$. Thus $C(n)$ is $O(n \log n)$. □

If no further restrictions are made on the nature of the path shrinking, then this bound is tight. A sequence Σ with cost $\Omega(n \log n)$ can be constructed as follows.

Let x denote the node with label $\lfloor n/2 \rfloor$. Let the initial tree be a simple path of n nodes. Then the first $\lfloor n/2 \rfloor$ finds are defined to be the finds in the most expensive sequence for a starting tree that consists of $\lfloor n/2 \rfloor$ nodes in a simple path, plus the addition of the edge update of x, which reassigns its parent to be its grandparent, if the grandparent exists. After the finds of the first $\lfloor n/2 \rfloor$ of the nodes, the tree has the shape of a simple path of $\lceil n/2 \rceil$ nodes. The remaining finds are of the form of the most expensive sequence of finds for a tree with $\lceil n/2 \rceil$ nodes.

Thus the total cost $L(n)$ of the sequence is at least $L(\lfloor n/2 \rfloor) + L(\lceil n/2 \rceil) + \lfloor n/2 \rfloor - 1$, or $\Omega(n \log n)$.

5. Conclusions. Unfortunately the results of this paper do not immediately translate into results concerning the splay tree and pairing heap data structures. Though much of their behavior is that of postorder path halving, it is not true that the rising roots condition holds.

We shall restrict our discussion to splay trees. The behavior of pairing heaps is strongly related to that of splay trees. The pairing done by the heap is similar to the pairing done by the splay tree during a sequence of zigzig rotations. What is now known concerning pairing heaps was obtained by using the same proof technique developed for the analysis of splay trees.

A splay operation on x rotates x to the root of the tree by repeatedly applying the appropriate case, i.e., zigzig, zigzag, or zig rotations [2]. The zigzig case is the crucial one, the case that separates splay trees from the many types of balanced trees [14], [15].

The sequential access problem is a special case of splaying in which every splay consists entirely of zigzig rotations, except possibly a final zigzag or zig rotation. In this problem the following operation is performed until the tree is empty. Splay on the least node in the symmetric order of the tree, then delete it (Fig. 2). This is precisely equivalent to doing postorder path halving in the general tree derived from T by the natural correspondence.

Another special case of splaying is when the splay tree is used to simulate a double ended queue, or deque [8]. That is, the only permitted operations are: splay on the least node in symmetric order, then delete it, or splay on the greatest node in symmetric order, then delete it.

The cost of such a sequence is dominated by the cost of the following problem [3]. Repeatedly do a partial splay on the least node in symmetric order, and then delete that node. A partial splay operation on x is identical to performing a splay operation on x that is allowed to terminate any time at or before the time when x reaches the root of the tree. That is, x is rotated higher and higher in the tree, using the appropriate zigzig or zigzag case, until you choose to terminate the operation, or a zig operation is performed, or x reaches the root.

The binary tree used by the partial splaying problem is derived from the binary tree in the original deque problem as follows. Rotate the edge between the root and its right child until the root has no right child. Thus a splay from the least node in the deque problem becomes a partial splay in the new problem.

This transformation of the tree changes the problem from doing complete splays in symmetric order to doing partial splays in symmetric order. This is necessary because in the original deque problem splays on the greatest node are allowed, and such a splay can rotate the edge between the root and its right child.

Our derived problem, that of doing partial splays in symmetric order, is equivalent to doing partial path halving in the postorder. Unfortunately the rising roots condition need not hold for this sequence of partial path halvings. In the binary tree corresponding to the deque problem, a splay from the least node can cause the left child of the root to be rotated over the root, becoming the new root. Thus the next splay, and the corresponding find, has as its root a node that is less than the previous root with respect to the symmetric order of the binary tree or the postorder of the general tree. This violates the rising roots condition.

Postorder path compression is also related to the study of dynamically optimal binary search tree algorithms. Sleator and Tarjan have conjectured that the splay tree algorithm is dynamically optimal [2]. In [16], we propose an off-line algorithm that we conjecture is also dynamically optimal. The cost of this algorithm, for the special case when the operations are only deque outputs, is dominated by the cost of postorder path compression when the rising roots condition does not hold.

This leaves us with the following open questions. What is the bound for any sequence of partial path compressions done in postorder? What is it for postorder path halving? Currently the best bounds are $O(n\alpha(n))$ as shown in [1] and [9]. What can be said about other path shrinking strategies? Path shrinking is so general an operation that the notion of the root of the find path is almost meaningless, and can be considered the root of the tree for every find. Thus the distinction between sequences that satisfy the rising roots condition and those that do not is not interesting here.

What condition must be imposed on the parent updating strategy to ensure that the total cost is linear? We conjecture that some kind of *proximity* condition is sufficient. That is, a rule of the form: for some fixed constant c, every subset of c consecutive nodes of the find path has at least one node whose parent pointer is updated.

REFERENCES

[1] S. HART AND M. SHARIR, *Nonlinearity of Davenport-Schinzel sequences and of generalized path compression schemes*, Combinatorica, 6 (1986), pp. 151-177.

[2] D. D. SLEATOR AND R. E. TARJAN, *Self-adjusting binary search trees*, J. Assoc. Comput. Mach., 32 (1985), pp. 652-686.

[3] J. LUCAS, *Arbitrary splitting in splay trees*, Tech. Report 234, Department of Computer Science, Rutgers University, New Brunswick, NJ, June 1988.

[4] M. L. FREDMAN, R. SEDGEWICK, D. D. SLEATOR, AND R. E. TARJAN, *The pairing heap, a new form of self-adjusting heap*, Algorithmica, 1 (1986), pp. 111-129.

[5] R. E. TARJAN AND J. VAN LEEUWEN, *Worst-case analysis of set union algorithms*, J. Assoc. Comput. Mach., 31 (1984), pp. 245-281.

[6] R. E. TARJAN, *Efficiency of a good but not linear set union algorithm*, J. Assoc. Comput. Mach., (1975), pp. 215-225.

[7] M. J. FISCHER, *Efficiency of equivalence algorithms*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, 1972, pp. 153-168.

[8] R. E. TARJAN, *Sequential access in splay trees takes linear time*, Combinatorica, 5 (1985), pp. 367-378.

[9] R. SUNDAR, *Twists, turns, cascades, deque conjecture, and scanning theorem*, in Proc. 30th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1989.

[10] M. LOEBL AND J. NESETRIL, *Linearity and unprovability of set union problem strategies*, in Proc. 20th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1988, pp. 360-366.

[11] J. NESETRIL, private communication, June 1988.

[12] M. LOEBL AND J. NESETRIL, *Linearity and unprovability of set union problem strategies* I. *Linearity of on-line postorder*, Tech. Report 89-04, Department of Computer Science, University of Chicago, Chicago, IL, April, 1989.

[13] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974, pp. 135-139.

[14] G. M. ADELSON-VELSKII AND E. M. LANDIS, *An algorithm for the organization of information*, Sov. Math. Dokl., 3 (1962), pp. 1259-1262.

[15] L. J. GUIBAS AND R. SEDGEWICK, *A dichromatic framework for balanced trees*, in Proc. 10th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1978, pp. 8-21.

[16] J. LUCAS, *Canonical forms for competitive binary search tree algorithms*, Tech. Report 250, Department of Computer Science, Rutgers University, New Brunswick, NJ, December 1988.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 883-901, October 1990

008

A SEMIRING ON CONVEX POLYGONS AND ZERO-SUM CYCLE PROBLEMS*

KAZUO IWANO† AND KENNETH STEIGLITZ‡

Abstract. Two natural operations on the set of convex polygons are shown to form a closed semiring; the two operations are vector summation and convex hull of the union. Various properties of these operations are investigated. Kleene's algorithm applied to this closed semiring solves the problem of determining whether a directed graph with two-dimensional labels has a zero-sum cycle or not. This algorithm is shown to run in polynomial time in the special cases of graphs with one-dimensional labels, BTTSP (Backedged Two-Terminal Series-Parallel) graphs, and graphs with bounded labels. The undirected zero-sum cycle problem and the zero-sum *simple* cycle problem are also investigated.

Key words. semiring, convex polygon, dynamic graph, algorithm, complexity

AMS(MOS) subject classifications. 05, 13, 16, 52, 68

1. Introduction.

In this paper, we show that two natural operations on the set of convex polygons form a closed semiring; the two operations are vector summation and convex hull of the union. We then investigate the time complexity of each operation and its effect on the number of edges of the polygons.

Kleene's algorithm applied to various closed semirings leads to efficient algorithms for a variety of problems; for example, finding the shortest paths for all pairs of nodes [3], converting a finite automaton into a regular expression, and finding the most reliable or largest-capacity path [5]. In this paper we use the above closed semiring to solve the *zero-sum cycle problem* in doubly weighted directed graphs.

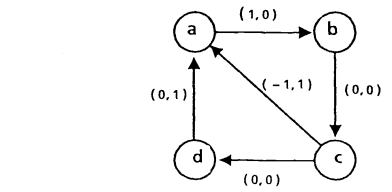
Doubly weighted graphs, which have a two-dimensional weight on each edge, have been studied by Lawler [19], Dantzig, Blattner, and Kao [7], and Reiter [24]. The cost of a path is defined as the sum of weights of edges on the path. The *doubly weighted zero-sum cycle problem* is to find a cycle whose cost in each dimension is 0. In [12], [13], [14], [15], [17], we saw that certain problems in VLSI applications involving a regular structure can be transformed to problems in two-dimensional infinite graphs consisting of repeated finite graphs. Repeated use of a doubly weighted digraph, called the *static graph* G^0, forms a *dynamic graph* G^2. As shown in Fig. 1, each label of the static graph G^0 indicates the differences between the x- and y-coordinates of two connected vertices in G^2. The absence of a zero-sum cycle in the specified static graph is then necessary and sufficient for the acyclicity of the associated dynamic graph. If a two-dimensional regular electrical circuit is associated with a dynamic graph, acyclicity of the dynamic graph implies that the associated electrical circuit is free of an electrical "short circuit" [12].

Since the cost of each path between any two vertices can be regarded as a point in the two-dimensional Euclidean plane, we can associate a pair of vertices v and w with a convex polygon α_{vw} as follows: α_{vw} is the convex hull of all points associated

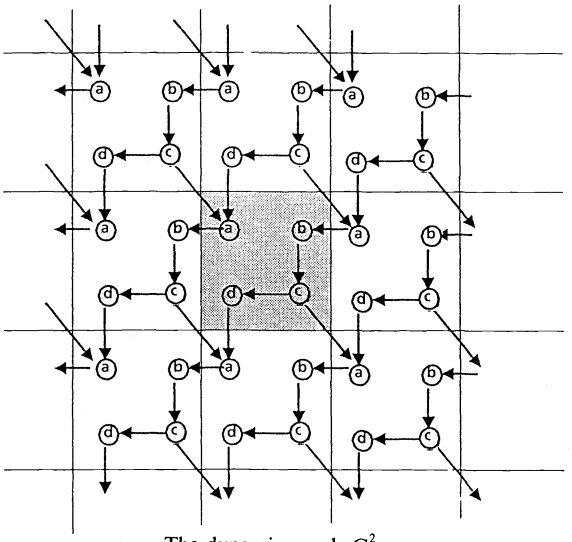
* Received by the editors September 15, 1986; accepted for publication (in revised form) February 6, 1990. This work was supported by National Science Foundation grant ECS-8307955; U.S. Army Research Office, Durham, North Carolina, under grant DAAG29-82-K-0095; Defense Advanced Research Projects Agency contract N00014-82-K-0549; and IBM-Japan. These results appeared in preliminary form in Proceedings of the 19th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, May 25-27, 1987, pp. 46-55. The original title was "Testing for cycles in infinite graphs with periodic structure."

† IBM Research, Tokyo Research Laboratory, Sanbancho, Chiyodaku, Tokyo 102, Japan.

‡ Department of Computer Science, Princeton University, Princeton, New Jersey 08544.



A static graph G^0



The dynamic graph G^2

FIG. 1. *A static graph G^0 shows how to connect nodes in G^2. The shaded area shows a basic cell.*

with costs of paths from v to w. We apply the two operations above to the set of these convex hulls, and use the closed semiring defined by these two operations to solve the doubly weighted zero-sum cycle problem. We show that this algorithm runs in polynomial time in the special cases of bounded label graphs, BTTSP graphs (the Backedged Two-Terminal Series-Parallel graphs), and graphs with one-dimensional labels. The 1-*bounded graphs*, whose labels are 0, 1, or -1, arise in VLSI applications where the interconnections between regular basic cells are made locally. The BTTSP graphs are an extension of the class of *Two-Terminal Series-Parallel* [1], [8], [26], [27]. When the extended abstract of the present paper appeared in [16], the question of whether the zero-sum cycle problem for general graphs is in P remained open. Kosaraju and Sullivan [18] subsequently showed that the zero-sum cycle problem for any dimension can be formulated in terms of linear programming and is thus solvable in polynomial-time. Recently Cohen and Megiddo [6] proved that the zero-sum cycle problem for any fixed dimension belongs to the class NC, and can be solved in the two-dimensional case in serial time $O(nm)$, the best result to date. We hope the present paper retains independent interest as a new connection between convex polygons and semirings, and as a novel application of Kleene's closure algorithm.

Finally, we discuss variations of the zero-sum cycle problem, the undirected case, and the zero-sum *simple* cycle problem.

2. Two operations and a semiring. We define our closed semiring [21] as follows: Let S be the set of all convex polygons whose vertices have integer coordinates. That

is, $S=\{\alpha^{\cup} \mid \alpha \in 2^{Z\times Z}\}$, where $\alpha^{\cup}$ indicates the convex hull of α. Notice that this definition allows polytopes with an infinite number of edges, unbounded area, or zero area, but does not allow curves. Thus our usage of the term *convex polygon* is more general than the conventional one. We conventionally denote an element in S by a lowercase Greek letter. We regard a point or a line segment as an element of S.

For any two sets $\alpha, \beta \in S$, we define the new set called a *vector sum of* α and β as follows: $\alpha * \beta = \{(x, y) \mid$ there exist elements $(a_x, a_y) \in \alpha$ and $(b_x, b_y) \in \beta$ such that $x = a_x + b_x,\ y = a_y + b_y\}$. See [11] for details of this operation. Let $\mathbf{0} = \{(0, 0)\} \in S$ and $\varnothing$ be the empty set.

We define the $\uplus$ operation as the convex hull of the union of two convex polygons in S; that is, $\alpha \uplus \beta = (\alpha \cup \beta)^{\cup}$ for any $\alpha, \beta \in S$. In this paper, we call the $\uplus$ operation *union-sum*. We can naturally define a union-sum of a countable number of convex polygons as follows: Let I be a countable (finite or infinite) index set and $\alpha_i \in S$ for all $i \in I$. Then we define *union-sum* $\uplus_{i\in I} \alpha_i$ by $\uplus_{i\in I} \alpha_i = (\cup_{i\in I} \alpha_i)^{\cup}$. Since $\cup_{i\in I} \alpha_i$ exists and is unique, its convex hull $\uplus_{i\in I} \alpha_i$ exists and is unique. Note that α_i is the convex hull of some set in $2^{Z\times Z}$, and thus every vertex of $\cup_{i\in I} \alpha_i$ is in $2^{Z\times Z}$. Hence $\uplus_{i\in I} \alpha_i \in S$, and thus the union-sum above is well defined.

We now define the $+$ operation as the convex hull of the vector summation of two convex polygons in S; that is, $\alpha + \beta = (\alpha * \beta)^{\cup}$. By convention, we define $\alpha + \varnothing = \varnothing + \alpha = \varnothing$. Note as we show later (Corollary 3.4, § 3), that $\alpha * \beta$ is itself a convex polygon when α and β are convex polygons. Therefore $\alpha + \beta = (\alpha * \beta)^{\cup} = \alpha * \beta$ for any $\alpha, \beta \in S$. Therefore, we identify $+$ with $*$, and call the $+$ operation *vector-sum*. From the definitions, the vector-sum operation is commutative. Fig. 2 shows an example of the vector-sum of two convex polygons.

We now have the following theorem.

THEOREM 2.1. *The system* $(S, \uplus, +, \varnothing, \mathbf{0})$ *is a closed semiring.*

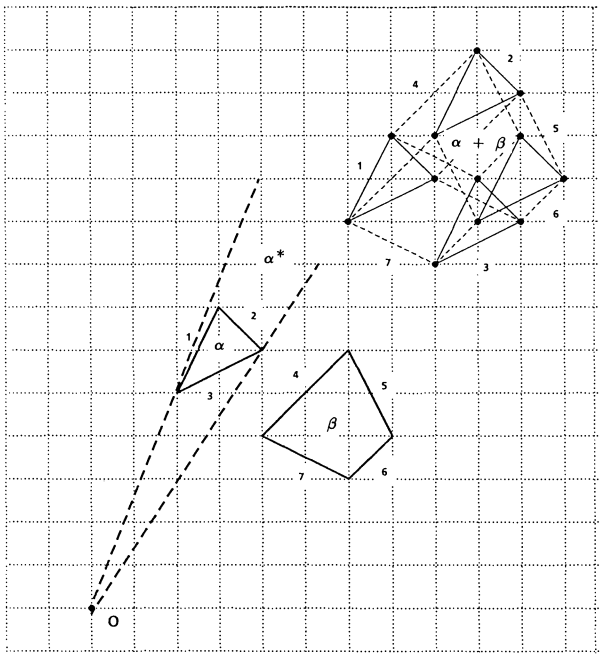


FIG. 2. $\alpha + \beta$ *is bounded by edges that are aligned with the edges in* α *or* β. *Aligned edges have the same numbers.*

Before proving this theorem, we need the following lemma.

LEMMA 2.2. *Let I be a countable index set. Let $\alpha_i \in 2^{Z\times Z}$ for all $i \in I$. Then we have* $(\cup_{i\in I} \alpha_i^{\cup})^{\cup} = (\cup_{i\in I} \alpha_i)^{\cup}$.

Proof. Let A be the left-hand side of the above equation and B the right-hand side. Since $\alpha_i \subset A$ for all $i \in I$, we have $\cup_{i\in I} \alpha_i \subset A$. Since A is a convex hull, we have $(\cup_{i\in I} \alpha_i)^{\cup} = B \subset A$.

Note that $\alpha_i \subset B$ and thus $\alpha_i^{\cup} \subset B$ for all $i \in I$. Therefore $\cup_{i\in I} (\alpha_i^{\cup}) \subset B$, and moreover, since B is a convex polygon, $A = (\cup_{i\in I} \alpha_i^{\cup})^{\cup} \subset B$. □

Now we prove Theorem 2.1.

Proof of Theorem 2.1. We show that the system $(S, \uplus, +, \varnothing, \mathbf{0})$ satisfies the six properties of a closed semiring.

(1) $(S, \uplus, \varnothing)$ *is a commutative monoid.* This is immediate from the definition and Lemma 2.2.

(2) $(S, +, \mathbf{0})$ *is a monoid.* From the definition, this is trivial.

(3) $+$ *distributes over* $\uplus$. Let $\alpha, \beta, \gamma \in S$ be convex polygons. Since $\alpha + (\beta \uplus \gamma)$ is convex and contains $(\alpha+\beta)$ and $(\alpha+\gamma)$, we have $(\alpha+\beta) \uplus (\alpha+\gamma) \subset \alpha + (\beta \uplus \gamma)$.

For the opposite direction, let x be a point in $\alpha + (\beta \uplus \gamma)$. Then x can be expressed as $a + \lambda b + (1-\lambda)c$, where $\lambda \in [0, 1]$. Then we have $x = a + \lambda b + (1-\lambda)c = \lambda(a+b) + (1-\lambda)(a+c) \in (\alpha+\beta) \uplus (\alpha+\gamma)$. Therefore, $+$ distributes over $\uplus$. Note that we can also prove that $+$ distributes over finite union-sums by induction.

Since $\alpha \uplus \alpha = (\alpha \cup \alpha)^{\cup} = \alpha^{\cup} = \alpha$, $+$ is idempotent.

(4) Let $I = \{i_1, i_2, \cdots, i_k\}$ be a finite nonempty index set. Let $\alpha_i \in S$ for all $i \in I$. Then we can prove $\uplus_{i\in I} \alpha_i = \alpha_{i_1} \uplus \alpha_{i_2} \uplus \cdots \uplus \alpha_{i_k}$ by induction on k and Lemma 2.2. For the empty index set $I = \varnothing$, we have $\uplus_{i\in\varnothing} \alpha_i = \varnothing$.

(5) *The result of union-sum does not depend on the ordering of the factors.* The proof is straightforward from the definition of $\uplus$ and Lemma 2.2.

(6) In addition to (3), $+$ *distributes over countably infinite union-sums* $\uplus$. Let $\beta \in S$ and $\alpha_i \in S$ for $I = \{1, 2, \cdots\}$. Then we prove that $\beta + \uplus_{i\in I} \alpha_i = \uplus_{i\in I} (\alpha_i + \beta)$. Let $Z_\alpha = \uplus_{i\in I} \alpha_i$ and $Z_{\alpha+\beta} = \uplus_{i\in I} (\alpha_i + \beta)$. We first prove that $\beta + Z_\alpha \subset Z_{\alpha+\beta}$. Let $p = b + x$ be an arbitrary point in $\beta + Z_\alpha$ with $b \in \beta$ and $x \in Z_\alpha$. If there exists a finite set of indexes J such that $x \in \uplus_{j\in J} \alpha_j$, then from (3), $p = b + x \in \beta + \uplus_{j\in J} \alpha_j \in \uplus_{j\in J} (\beta + \alpha_j) \subset Z_{\alpha+\beta}$. If x is not in the union-sum of a finite number of α_i's, then x can be represented as the limit point of a sequence of points, each of which is in some α_i; that is, there exists a countably infinite set of indexes $J = \{j_1, j_2, \cdots, j_i, \cdots,\}$ such that $x = \lim_{i\to\infty} x_{j_i}$ where $x_{j_i} \in \alpha_{j_i}$. Then

$$p = b + x = b + \lim_{i\to\infty} x_{j_i} = \lim_{i\to\infty} (b + x_{j_i}) \in \lim_{i\to\infty} (\beta + \alpha_{j_i})$$

$$\subset \uplus_{j\in J} (\beta + \alpha_j) \subset \uplus_{i\in I} (\beta + \alpha_i) = Z_{\alpha+\beta}.$$

Therefore $\beta + Z_\alpha \subset Z_{\alpha+\beta}$.

The converse can be proved similarly, and thus multiplication distributes over infinite sums. □

Having established that the structure $(S, \uplus, +, \varnothing, \mathbf{0})$ is a closed semiring, we can apply Kleene's algorithm to solve certain problems related to paths in a graph [2], [21]. With this goal in mind we next investigate the basic properties of the operations $+$ and $\uplus$.

3. Some properties of the + and $\uplus$ operations. Before stating some properties, we need some definitions. For a convex polygon α in S, we denote its edge set by α_E and its vertex set by α_V. Let l be an edge of α or a line that does not intersect α. Then we regard l as an oriented line with respect to α and define its direction, denoted by $\theta_l(\alpha)$, in the range $0 \leqq \theta_e < 2\pi$ such that α lies on the right-hand side of l when we

traverse l in its positive direction. Unless specified, θ_e means $\theta_e(\alpha)$ for an edge $e \in \alpha_E$. We regard $e \in \alpha_E$ as a vector $\mathbf{e}$ with the direction of $\theta_e(\alpha)$. Let $\alpha_{vector} = \{\mathbf{e} \mid e \in \alpha_E\}$. By convention, we define the following special cases: When α is either a point or the entire plane, we regard α as a special symbol and define $\alpha_{vector} = \{\alpha\}$. When α is a line segment e, we define $\alpha_{vector} = \{\mathbf{e}, -\mathbf{e}\}$.

Let $A = \{\alpha_i\}_{i \in I}$ be a set of convex polygons. We define $|A| = |\cup_{i \in I} (\alpha_i)_{vector}|$, that is, $|A|$ is the number of distinct vectors in $\cup_{\alpha_i \in A} (\alpha_i)_{vector}$. We also write $|A| = |\alpha|$ when A has the single element α. We say that two edges $e \in \alpha_E$ and $f \in \beta_E$ are *aligned* when $\theta_e(\alpha) = \theta_f(\beta)$.

Now we have the following lemma about the relationship between two consecutive edges of a convex polygon and their directi ns.

LEMMA 3.1. *Let e and f be two consecutive edges of a convex polygon α in clockwise order. Then $\theta_f < \theta_e$ or $\theta_e + \pi < \theta_f$.*

Proof. From the definition, f lies in the right-half plane of e. □

COROLLARY 3.2. *Let $\alpha_E = \{e_1, e_2, \cdots, e_m\}$ be the edges of a convex polygon α in clockwise order. Let θ_{e_1} be the maximum of $\{\theta_{e_i}\}$. Then $\theta_{e_1} > \theta_{e_2} > \cdots > \theta_{e_m}$. The set α_E is called the edge sequence when the elements of α_E are ordered as above.*

Proof. The proof is clear from Lemma 3.1. □

To analyze how the + operation affects the number of distinct vectors, we will use the following well-known theorem.

THEOREM 3.3 ([11]). *Let α, β be two convex polygons in S. Then for every $e \in \alpha_E \cup \beta_E$, there exists an edge $f \in (\alpha + \beta)_E$ that is aligned with e; that is, $\theta_f = \theta_e$. This enables us to define a function $f = \varphi(e)$ from $\alpha_E \cup \beta_E$ to $(\alpha + \beta)_E$. Moreover, the function φ is onto. Figure 2 illustrates this theorem.* □

COROLLARY 3.4. *Let α, β be convex polygons. Then $\alpha + \beta = \alpha * \beta = \beta * \alpha = \beta + \alpha$.*

Proof. For the proof see [11], [20], [28]. □

COROLLARY 3.5. *Let α and β be convex polygons in S such that both have a finite number of edges and $n = |\alpha + \beta|$. Then the edge sequence of $\alpha + \beta$ can be computed in $O(n)$ steps from two edge sequences α_E and β_E.*

Proof. From Theorem 3.3, every edge e in $\alpha + \beta$ has an associated edge f in $\alpha_E \cup \beta_E$ such that $\theta_f = \theta_e$. Thus the edge sequence of $(\alpha + \beta)_E$ can be obtained by merging the two sets $\{\theta_e \mid e \in \alpha_E\}$ and $\{\theta_e \mid e \in \beta_E\}$. □

COROLLARY 3.6. *Let $\alpha_1, \alpha_2, \cdots$, and α_n be convex polygons in S. Then we have an onto function φ from $(\alpha_1)_E \cup (\alpha_2)_E \cup \cdots \cup (\alpha_n)_E$ to $(\alpha_1 + \alpha_2 + \cdots + \alpha_n)_E$ such that $\theta_{\varphi(e)} = \theta_e$ for any $e \in (\alpha_1)_E \cup (\alpha_2)_E \cup \cdots \cup (\alpha_n)_E$.*

Proof. The proof is by induction on n and Theorem 3.3. □

THEOREM 3.7. *For any $\alpha, \beta \in S$, we have $|\alpha + \beta| \leqq |\{\alpha, \beta\}| \leqq |\alpha| + |\beta|$.*

Proof. The proof is straightforward from Theorem 3.3. □

Next we analyze the effect of the $\uplus$ operation on the number of distinct vectors. First we have the following theorem.

THEOREM 3.8. *Let α and β be bounded convex polygons in S. Then $|\alpha \uplus \beta| \leqq |\alpha| + |\beta|$.*

Before proving Theorem 3.8, we need the following lemma.

LEMMA 3.9. *Let $\alpha \in S$ and $p_1, p_2, \cdots, p_n$ be points. Then*

$$|\alpha \uplus p_1 \uplus p_2 \uplus \cdots \uplus p_n| \leqq |\alpha| + n.$$

Proof. This can be proved by induction on n. Suppose $n = 1$. If α contains p_1, then $|\alpha \uplus p_1| = |\alpha|$. Otherwise, $\alpha \uplus p_1$ contains at least one edge of α, and thus $|\alpha \uplus p_1| \leqq |\alpha| + 1$. Suppose the lemma holds for numbers less then n. Let $\beta_k = \alpha \uplus p_1 \uplus p_2 \uplus \cdots \uplus p_k$. Then from the induction hypothesis, $|\beta_{n-1}| \leqq |\alpha| + (n-1)$, and thus $|\beta_n| \leqq |\beta_{n-1}| + 1 \leqq |\alpha| + n$. □

Proof of Theorem 3.8. Note that $\alpha \uplus \beta$ is $\alpha \uplus p_1 \uplus p_2 \uplus \cdots \uplus p_n$ where $\beta_V = \{p_1, p_2, \cdots, p_n\}$. Thus from Lemma 3.9, $|\alpha \uplus \beta| \leqq |\alpha| + n = |\alpha| + |\beta|$. □

Theorem 3.11 covers the case when α or β is an unbounded convex polygon α^*, which is defined as follows: For a convex polygon α and a nonnegative integer n, we define a convex polygon α^n as follows: (1) $\alpha^0 = \mathbf{0}$ and (2) $\alpha^n = \alpha + \alpha^{n-1}$ for $n > 1$. Since a system $(S, \uplus, +, \varnothing, \mathbf{0})$ is a closed semiring, we can define the convex polygon α^* by $\alpha^0 \uplus \alpha^1 \uplus \cdots = \uplus_{i=0}^{\infty} \alpha^i$. As shown in Fig. 2, α^* is $\cup_{p \in \alpha} (\cup_{\lambda > 0} \lambda p)$. Thus α^* is essentially a cone emanating from the origin. As a special case, α^* may be the entire plane, a half plane, a line, a half line, or the origin itself. Now we analyze the effect of the * operation on the number of distinct vectors.

LEMMA 3.10. *For two convex polygons α and γ, we have* $|\alpha + \gamma^*| \leqq |\alpha| + 1$.

Proof. If γ^* is either the entire plane, a half plane, a line, a half line, or the origin itself, the proof is straightforward. Otherwise γ^* is a cone emanating from the origin and has two edges g_1' and g_2'. Let g_1 (respectively, g_2) be the support lines at v (respectively, w) of the convex polygon α such that $\theta_{g_1}(\alpha) = \theta_{g_1'}(\gamma^*)$ and $\theta_{g_2}(\alpha) = \theta_{g_2'}(\gamma^*)$. If $v = w$, then $|\alpha + \gamma^*| = |\gamma^*| \leqq 2$. If $v \neq w$, then as shown in Fig. 3, there must be at least one edge of α which is inside $\alpha + \gamma^*$. Thus the lemma is proved. □

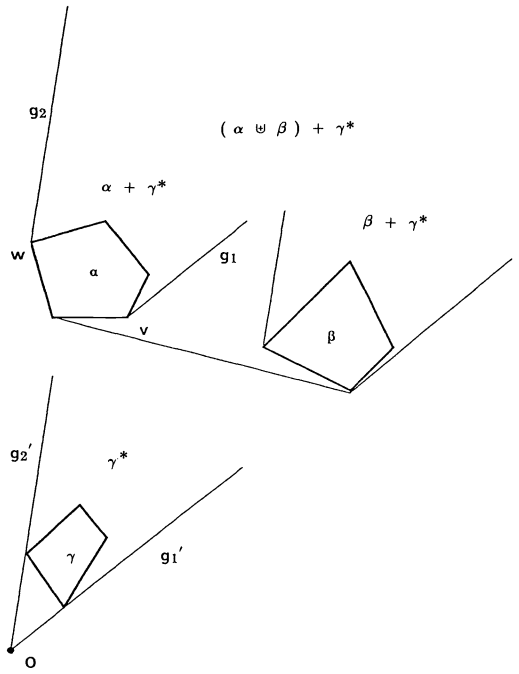


FIG. 3. $|\alpha + \gamma^*| \leqq |\alpha| + 1$ *and* $\alpha \uplus (\beta + \gamma^*) = (\alpha \uplus \beta) + \gamma^*$.

The above lemma shows that replacement of α by $\alpha + \gamma^*$ does not increase the number of edges by more than one. Moreover, we have a stronger result in the following theorem, which shows the same result for any number of such replacements in a series of $\uplus$ operations. We will use this theorem in §§ 5 and 6.

THEOREM 3.11. *Let $\beta_i, \gamma_i \in S$ for $i = 1, 2, \cdots, n$. Then we have*

$$|(\beta_1 + \gamma_1^*) \uplus (\beta_2 + \gamma_2^*) \uplus \cdots \uplus (\beta_n + \gamma_n^*)| \leqq |\beta_1 \uplus \beta_2 \uplus \cdots \uplus \beta_n| + 1.$$

Before proving Theorem 3.11, we need some lemmas.

LEMMA 3.12. *Let $\gamma_i \in S$ for $i = 1, 2, \cdots, n$. Then*

$$\gamma_1^* + \gamma_2^* + \cdots + \gamma_n^* = (\gamma_1 \uplus \gamma_2 \uplus \cdots \uplus \gamma_n)^*.$$

Proof. We prove this by induction on n, using k for the index of induction. The lemma trivially holds for $k = 1$. When $k = 2$, we prove that $\gamma_1^* + \gamma_2^* = (\gamma_1 \uplus \gamma_2)^*$. Since $\gamma_1^*, \gamma_2^* \subset (\gamma_1 \uplus \gamma_2)^*$, we have $\gamma_1^* + \gamma_2^* \subset (\gamma_1 \uplus \gamma_2)^*$. We next prove the opposite direction. Since $\mathbf{0} \in \gamma_1^* + \gamma_2^*$ we have from the distributive law $\gamma_1 \uplus \gamma_2 \subset (\gamma_1 \uplus \gamma_2) + (\gamma_1^* + \gamma_2^*) = (\gamma_1 + \gamma_1^* + \gamma_2^*) \uplus (\gamma_1 + \gamma_2 + \gamma_2^*) \subset \gamma_1^* + \gamma_2^*$. Thus the lemma holds for $k = 2$.

Assume that the lemma holds for $k < n$, then $\gamma_1^* + \gamma_2^* + \cdots + \gamma_n^* = (\gamma_1 \uplus \gamma_2 \uplus \cdots \uplus \gamma_{n-1})^* + \gamma_n^* = (\gamma_1 \uplus \gamma_2 \uplus \cdots \uplus \gamma_n)^*$. Note that we used the case $k = n-1$ for the first transformation and $k = 2$ for the latter. □

LEMMA 3.13. *Let α, β, and γ be convex polygons. Then*

$$\alpha \uplus (\beta + \gamma^*) = (\alpha \uplus \beta) + \gamma^*.$$

Proof. For the proof see Fig. 3. Since $\alpha \subset \alpha + \gamma^*$, we have

$$\alpha \uplus (\beta + \gamma^*) \subset (\alpha + \gamma^*) \uplus (\beta + \gamma^*) = (\alpha \uplus \beta) + \gamma^*.$$

We now prove the opposite direction; that is, $(\alpha \uplus \beta) + \gamma^* \subset \alpha \uplus (\beta + \gamma^*)$. Since $\beta + \gamma^* \subset \alpha \uplus (\beta + \gamma^*)$, we only have to prove that $\alpha + \gamma^* \subset \alpha \uplus (\beta + \gamma^*)$. Let $p = a + g$ be a point in $\alpha + \gamma^*$ with $a \in \alpha$ and $g \in \gamma^*$. Let b be an arbitrary point in β. Let p_n be a point obtained by the following equation when we regard p_n, a, b, and g as points in the $x-y$ plane: $p_n = (1 - 1/n)a + (1/n)(b + ng)$. Then p_n is on the line segment $a, (b + g^n)$, and thus $p_n \in \alpha \uplus (\beta + \gamma^*)$. Note that $p_\infty = \lim_{n \to \infty} p_n$ is also in $\alpha \uplus (\beta + \gamma^*)$ and $p_\infty = a + g = p$. Therefore $\alpha + \gamma^* \subset \alpha \uplus (\beta + \gamma^*)$. □

LEMMA 3.14. *Let $\alpha_i, \gamma_i \in S$ for $i = 1, 2, \cdots, n$. Then*

$$(\alpha_1 + \gamma_1^*) \uplus (\alpha_2 + \gamma_2^*) \uplus \cdots \uplus (\alpha_n + \gamma_n^*)$$
$$= (\alpha_1 \uplus \alpha_2 \uplus \cdots \uplus \alpha_n) + \gamma_1^* + \gamma_2^* + \cdots + \gamma_n^*.$$

Proof. Denote the left-hand side of the above equation by A_n, and the right-hand side by B_n. We prove this by induction on n and use k for the index of induction. The lemma holds trivially for $k = 1$. When $k = 2$, from Lemma 3.13, $A_2 = ((\alpha_1 + \gamma_1^*) \uplus \alpha_2) + \gamma_2^* = (\alpha_1 \uplus \alpha_2) + \gamma_1^* + \gamma_2^* = B_2$. Assume that the lemma holds for $k < n$. From the induction hypothesis for $k = n-1$, $A_n = B_{n-1} \uplus (\alpha_n + \gamma_n^*)$. We then obtain $A_n = ((\alpha_1 \uplus \alpha_2 \uplus \cdots \uplus \alpha_{n-1}) + (\gamma_1 \uplus \gamma_2 \uplus \cdots \uplus \gamma_{n-1})^*) \uplus (\alpha_n + \gamma_n^*)$ by applying Lemma 3.12 to B_{n-1}. From the basis of the induction ($k = 2$), $A_n = (\alpha_1 \uplus \alpha_2 \uplus \cdots \uplus \alpha_n) + (\gamma_1 \uplus \gamma_2 \uplus \cdots \uplus \gamma_{n-1})^* + \gamma_n^*$. From Lemma 3.12, we get $A_n = B_n$. □

We can now prove Theorem 3.11.

Proof of Theorem 3.11. From Lemma 3.12 and 3.14, we have

$$(\beta_1 + \gamma_1^*) \uplus (\beta_2 + \gamma_2^*) \uplus \cdots \uplus (\beta_n + \gamma_n^*)$$
$$= (\beta_1 \uplus \beta_2 \uplus \cdots \uplus \beta_n) + \gamma_1^* + \gamma_2^* + \cdots + \gamma_n^*$$
$$= (\beta_1 \uplus \beta_2 \uplus \cdots \uplus \beta_n) + (\gamma_1 \uplus \gamma_2 \uplus \cdots \uplus \gamma_n)^*.$$

Let A (respectively B) be the left- (respectively right-)hand side of the equation in the theorem. From Lemma 3.10, $|A| \leqq |\beta_1 \uplus \beta_2 \uplus \cdots \uplus \beta_n)| + 1 = |B| + 1$. □

THEOREM 3.15. *Let $|\alpha| + |\beta| = n$. The operations $+$, $\uplus$, and $*$ can all be done in $O(n)$ steps.*

Proof. We assume that two edge sequences $(\alpha)_E$ and $(\beta)_E$ are available. From Corollary 3.5, we know that the + operation takes $O(n)$ time. Given the edge-sequences of two convex polygons, the convex hull can be found in $O(n)$ time. There is also an algorithm in [4] that computes $l \cap \alpha$ in $O(\log(|\alpha|))$ steps where l is a line segment and α is a convex hull. Therefore the * operation takes $O(\log(|\alpha|))$ time. □

4. Application of the closed semiring. In this section, we define the doubly weighted zero-sum cycle problem and solve it by using the closed semiring defined in § 2.

Our instance is a doubly weighted digraph $G=(V, E, T)$ where V is its vertex set, E is its edge set, and T is a two-dimensional labeling such that $T(e)=(e_x, e_y) \in Z \times Z$ for every $e \in E$. We use n (respectively, m) to denote the number of vertices (respectively, edges) in a graph. We also use $\mathbf{0}$ to denote $(0, 0)$. A path P in G is a sequence of vertices $P=v_0, v_1, \cdots, v_k$ where $e_i=(v_{i-1}, v_i) \in E$ and $v_i \in V$. If all vertices $v_0, v_1, \cdots, v_{k-1}$ are distinct, a path P is *simple*. A path P such that $v_0=v_k$ is called a *cycle*. Given a path $P=v_0, v_1, \cdots, v_k$, we define the *cost of the path* $T(P)$ by the component-wise summation of edge-labels on that path; that is, $T(P)=\sum_{i=1}^{k} T(e_i)=(\sum_{i=1}^{k} e_{i_x}, \sum_{i=1}^{k} e_{i_y})$. A cycle W with $T(W)=\mathbf{0}$ is called a *zero-sum cycle*. We can now define the *doubly weighted zero-sum cycle problem* as follows:

Problem ZSC. Doubly Weighted Zero-sum Cycle Problem.

Instance: A doubly weighted digraph $G=(V, E, T)$ where T is a two-dimensional labeling such that $T(e)=(e_x, e_y) \in Z \times Z$ for every $e \in E$.

Question: Does G have a zero-sum cycle? In other words, is there a cycle W such that $T(W)=\mathbf{0}$?

By using the fact that the two operations defined on convex polygons form a closed semiring, we can answer this question with the Floyd-Warshall algorithm [2], [3], [10], [23].

Algorithm ZSC.

Input: A doubly weighted graph G with $V=\{v_1, v_2, \cdots, v_n\}$.

Output: This algorithm answers "Yes" if the digraph G has a zero-sum cycle; otherwise the algorithm answers "No."

Method: Let $PATH(v_i, v_j, k)$ denote the set of all paths from v_i to v_j such that all vertices on the path, except possibly the endpoints, are in the set $\{v_1, v_2, \cdots, v_k\}$. We compute the convex hull α_{ij}^k for $1 \leqq i, j \leqq n$ and $0 \leqq k \leqq n$, which is the convex hull of costs of all paths in $PATH(v_i, v_j, k)$.

```
.   procedure zero-sum cycle
.     begin
1.      for 1 ≦ i, j ≦ n do α_ij^0 = { {T((v, w))}  if (v, w) ∈ E
                                     { ∅            otherwise.
2.      for k = 1 to n
.         do
3.           for 1 ≦ i, j ≦ n do
4.             α_ij^k = α_ij^{k-1} ⊎ (α_ik^{k-1} + (α_kk^{k-1})* + α_kj^{k-1});
5.           if ∃i ∈ {1, 2, ···, n} s.t. 0 ∈ α_ii^k
.              then exit ("Yes");
.         od
6.      exit ("No");
.     end
```

THEOREM 4.1. *Algorithm* ZSC *works correctly.*

Before proving Theorem 4.1, we need the following lemmas.

LEMMA 4.2. *If there is a zero-sum cycle W, there must be a vertex v_i such that $\mathbf{0} \in \alpha_{ii}^n$.*

Proof. Let v_i be a vertex that is on the cycle W. Since the convex hull α_{ii}^n includes all costs of paths from v_i to v_i, we have $\mathbf{0} \in \alpha_{ii}^n$. □

LEMMA 4.3. *If $\mathbf{0} \in \alpha_{ii}^n$, there must be a zero-sum cycle W, and the vertex v_i is on the cycle W.*

Proof. Suppose that $\mathbf{0} \in \alpha_{ii}^n$. Let $(\alpha_{ii}^n)_V = \{s_1, s_2, \cdots, s_l\}$ such that $s_j \in Z \times Z$. Since α_{ii}^n is a convex polygon, any point z in α_{ii}^n can be represented as $z = \sum_{s_j \in (\alpha_{ii}^n)_V} k_j s_j$ such that $k_j \geqq 0$. Let C_j be a cycle corresponding to s_j such that $T(C_j) = s_j$ and v_i is on the cycle C_j. Note that since every s_j has integral coordinates, k_j can be chosen rational, if $z \in Z \times Z$. Thus there are rational numbers k_j' such that $\mathbf{0} = \sum_{s_j \in (\alpha_{ii}^n)_V} k_j' s_j$. There is an integer K such that all $K \cdot k_j'$ are integers. Thus $K \cdot \mathbf{0} = \mathbf{0} = \sum_{s_j \in (\alpha_{ii}^n)_V} (K \cdot k_j) s_j$. Then the desired cycle W consists of $K \cdot k_j$ copies of C_j for $s_j \in (\alpha_{ii}^n)_V$. □

Now we prove Theorem 4.1.

Proof of Theorem 4.1. From Lemma 4.2 and 4.3, in order to find a zero-sum cycle, we only have to check whether or not there exists some i such that $\mathbf{0} \in \alpha_{ii}^n$. We can prove that α_{ij}^k is correctly computed by the algorithm by induction on k (as in [2]). □

THEOREM 4.4. *Algorithm* ZSC *uses $O(n^3)$ $\uplus$, $+$, and $*$ operations from the closed semiring defined above, where n is the number of vertices in G.*

Proof. Line 4 is executed n^3 times in total. □

5. Special cases of the zero-sum cycle problem. In this section, we discuss the special cases of the zero-sum cycle problem where (1) the graphs have one-dimensional labels, (2) the graphs are undirected, (3) the graphs have labels with magnitude at most M, and (4) we are looking for a *simple* cycle with zero-sum. The first three cases have low order polynomial algorithms, whereas the fourth is NP-complete.

(1) The one-dimensional zero-sum cycle problem. We can solve the problem efficiently in the one-dimensional case as follows.

THEOREM 5.1. *The-dimensional zero-sum cycle problem can be solved in $O(n^3)$ time, where n is the number of vertices.* (*This result is implicit in Orlin* [22].)

Proof. We can apply our algorithm ZSC by ignoring the second labels. Note that in the one-dimensional case, every α_{ij}^k has at most two vertices, since it is either a point, a line segment, or a line on the x-axis. Thus $|\alpha_{ij}^k| \leqq 2$. From Theorem 3.15, each operation $\uplus$, $+$, or $*$ takes constant time. Hence from Theorem 4.4, the algorithm ZSC takes $O(n^3)$ time. □

(2) The two-dimensional undirected zero-sum cycle problem. We assume that G is connected. We will show that the undirected version of the zero-sum cycle problem can be solved in $O(m \log m)$ time, where m is the number of edges. In the undirected case, a path can traverse an edge in either direction.

An instance of the undirected problem is as follows:

Instance: A connected undirected graph $G = (V, E)$ with $V = \{v_1, v_2, \cdots, v_n\}$ and $E = \{e_1, e_2, \cdots, e_m\}$. A two-dimensional labeling T from E to $Z \times Z$ with $T(e) = (e_x, e_y)$ for every $e \in E$.

Now we have the following lemma:

LEMMA 5.2. *Let G and T be defined above. Let H_G be the convex hull of $\{T(e) \mid e \in E\}$. A necessary and sufficient condition for the existence of a zero-sum cycle is that exactly one of the following two conditions holds*:

(1) *The convex polygon H_G properly contains the origin.*

(2) *The origin is on an edge h of the convex polygon H_G. Let $Y = \{e \in E \mid T(e)$ is on $h\}$. Then there exists an edge $e \in Y$ such that $T(e) = \mathbf{0}$, or there are two edges $e_1, e_2 \in Y$ such that e_1 and e_2 are adjacent in G and the origin is on the line segment $\overline{T(e_1), T(e_2)}$.*

Before proving Lemma 5.2, we need some definitions. Let $X=\{C_e \mid e \in E\}$ such that C_e is the cycle $v \to w \to v$ where $e=(v, w)$. Then $T(C_e)=2T(e)$.

We call a set of cycles $A=\{W_i \mid i \in I\}$ *nullable* if there exists a set of nonnegative integers $A_Z=\{n_i \in Z^+ \cup \{0\} \mid i \in I\}$ such that the n_i are not all 0 and $\sum_{i \in I} n_i T(W_i)=\mathbf{0}$. If $(\bigcup_{i \in I} W_i)$ is connected, we say that A is *connected.*

Note that we can construct a zero-sum cycle from a connected nullable set. Now we have the following lemmas.

LEMMA 5.3. *Let G, T, and X be defined as above. Let $A=\{W_i \mid i \in I\}$ be a nullable set of cycles. Then we can find a connected nullable set B.*

Proof. Since A is nullable, there exists a set of nonnegative integers $A_Z=\{n_i \in Z^+ \cup \{0\} \mid i \in I\}$ such that the n_i are not all 0 and $\sum_{i \in I} n_i T(W_i)=\mathbf{0}$. If A is connected, A is the desired set. Suppose A is not connected. Let v_i be an arbitrary point on W_i for every $i \in I$. Since G is connected, there is a cycle P_i that passes through v_1 and v_i for every $i \in I-\{1\}$. Let k be a large positive integer. Let Q_i be a cycle consisting of k copies of W_i and one copy of P_i for every $i \in I-\{1\}$. Let $Q_1=W_1$. Then

$$T(Q_i)=\begin{cases} kT(W_1) & \text{for } i=1, \\ kT(W_i)+T(P_i) & \text{for } i \in I-\{1\}. \end{cases}$$

Since the convex hull of $\{T(W_i) \mid i \in I\}$ contains $\mathbf{0}$, the convex hull of $\{T(Q_i) \mid i \in I\}$ contains $\mathbf{0}$ for some large k. Therefore $B=\{Q_i \mid i \in I\}$ is nullable for large k. Since $v_1 \in \bigcap_{i \in I} Q_i$, B is connected. Thus B is the desired set. □

Now we prove Lemma 5.2.

Proof of Lemma 5.2. Suppose (1) holds. Note that $T(C_e)=2T(e)$, where C_e is the cycle for every $e \in E$ defined as above. Since $A=\{2T(e) \mid e \in E\}$ is a nullable set, we can find a connected nullable set, by Lemma 5.3. Thus there is a zero-sum cycle in G. When (2) holds, it is obvious that there is a zero-sum cycle in G.

Conversely, suppose there exists a zero-sum cycle W. From the definition, there exist positive integers n_e for $e \in W$ such that $\sum_{e \in W} n_e T(e)=\mathbf{0}$. This means that the convex hull of $\{T(e) \mid e \in E\}$, denoted by H_G, contains the origin. If H_G contains the origin properly, (1) holds. Otherwise, there must be an edge $e \in E$ such that $T(e)=\mathbf{0}$, or the origin must be on an edge h of H_G. Now we assume that $T(e) \neq \mathbf{0}$ for every $e \in E$. Let $Y=\{e \in E \mid T(e)$ is on the edge $h\}$. Since W is nullable, every edge in W is in Y. Let $\tilde{e}$ be an edge in Y. Then for every edge $e \in Y$, there exists k_e and $T(e)=k_e T(\tilde{e})$. Let $W_+=\{e \in W \mid T(e)=k_e T(\tilde{e}), k_e>0\}$, and let $W_-=\{e \in W \mid T(e)=-k_e T(\tilde{e}), k_e>0\}$. From the definition of $\{n_e\}$, we have $\sum_{e \in W} n_e T(e)=(\sum_{e \in W_+} n_e k_e-\sum_{e \in W_-} n_e k_e) T(\tilde{e})=\mathbf{0}$. Note that $W_+ \neq \varnothing$ and $W_- \neq \varnothing$. Since $W=W_+ \cup W_-$ is connected, there must be connected edges $e_1 \in W_+$ and $e_2 \in W_-$. Thus (2) holds. □

THEOREM 5.4. *The two-dimensional undirected zero-sum cycle problem can be solved in $O(m \log m)$ time, where m is the number of edges.*

Proof. We only have to check condition (1) and (2) in Lemma 5.2, which can be done in $O(m \log m)$ time. □

(3) Graphs with bounded labels. A doubly weighted digraph $G=(V, E, T)$ is called an *M-bounded graph* if each dimension of every label is an integer in $[-M, M]$.

In many VLSI applications, the communication between regular cells is made locally: that is, interconnections are made only to neighbors. For example, $n \times n$ multipliers can be constructed from arrays of one-bit full adders with carry and sum signal connections to the neighbors of each cell [12], [13], [14], [15]. Parallel adders can also be constructed from one-bit full adders with carry connections to the neighbor of each cell [13]. Many systolic arrays are also implemented with interconnections to

neighbors. In such VLSI applications, the associated static digraphs of the regular structures are all 1-bounded graphs [12].

We have the following lemma about the number of edges of a convex polygon included in a bounded region.

LEMMA 5.5. *Let R be a rectangle of width w and height h. Let H be an arbitrary convex polygon included in R. Then* $|H| \leqq 2 \max(w, h) + 2$.

Proof. Without loss of generality, we can assume that $\max(w, h) = w$. Let H_u be the set of edges in H from its highest leftmost vertex to its highest rightmost vertex in clockwise order. When we traverse an edge in H_u, we move at least one unit in the x-direction. Thus the number of edges in H_u is at most w. There are at most two vertical edges in H. Thus $|H| \leqq 2 \max(w, h) + 2$. □

LEMMA 5.6. *Let G be an M-bounded graph with n vertices, then we have* $|\alpha_{ij}^k| \leqq 4nM + 3$.

Proof. Let β_{ij}^k be the convex hull of the costs of all simple paths in $PATH(v_i, v_j, k)$ (see the previous section for the definition). Note that the length of a simple path is at most nM in each dimension. Thus β_{ij}^k is bounded by the rectangle $[-nM, nM] \times [-nM, nM]$. Therefore, from Lemma 5.5, $|\beta_{ij}^k| \leqq 2 \cdot (2nM) + 2 = 4nM + 2$. From Theorem 3.11, $|\alpha_{ij}^k| \leqq |\beta_{ij}^k| + 1 \leqq 4nM + 3$. □

THEOREM 5.7. *The algorithm* ZSC *takes $O(n^4M)$ time for M-bounded graphs with n vertices.*

Proof. From Theorem 4.4 and Lemma 5.6, the algorithm ZSC takes $O(n^3 \cdot nM) = O(n^4M)$ time. □

(4) The zero-sum simple cycle problem.

THEOREM 5.8. *The zero-sum simple cycle problem* (ZSSC) *is* NP*-complete.*

Proof. Here we use a variant of the reduction from the subset sum to the directed path problem in the one-dimensional dynamic graphs discussed in [22]. It is obvious that ZSSC is in NP. We use reduction from the subset sum problem SS to ZSSC, where the problem SS is defined as follows:

Input: $\{a_i \in Z^+ \mid i \in I\}$ where $I = \{1, 2, \cdots, n\}$ and $B \in Z^+$.

Question: Is there a subset J of I such that $\sum_{j \in J} a_j = B$?

Given an instance I_{SS} of SS, we construct an instance I_{ZSSC} of the zero-sum simple cycle problem as follows: A directed graph $G = (V, E)$ is shown in Fig. 4 where

$$V = \{v_1, v_2, \cdots, v_n, w_1, w_2, \cdots, w_n\},$$

$$E = \{e_i = (v_{i-1}, v_i) \mid i = 1, 2, \cdots, n\}$$

$$\cup \{f_i = (v_{i-1}, w_i) \mid i = 1, 2, \cdots, n\}$$

$$\cup \{g_i = (w_i, v_i) \mid i = 1, 2, \cdots, n\}$$

$$\cup \{e_0 = (v_n, v_1)\}.$$

Let T be a two-dimensional labeling from E to $Z \times Z$ as follows:

$$\begin{cases} T(e_0) = (-B, 0), & \\ T(e_i) = T(g_i) = (0, 0) & \text{for } i = 1, 2, \cdots, n, \\ T(f_i) = (a_i, 0) & \text{for } i = 1, 2, \cdots, n. \end{cases}$$

Suppose I_{SS} has a solution J such that $\sum_{j \in J} a_j = B$. Then I_{ZSSC} has a solution of a simple cycle consisting of e_0, f_j and g_j for $j \in J$, and e_i for $i \notin J$.

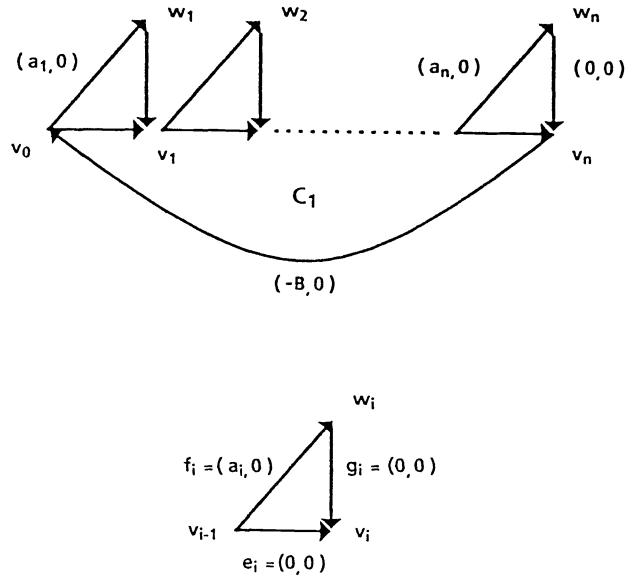


FIG. 4. *The graph above has a zero-sum cycle if and only if there exists a set* $J \subset I = \{1, 2, \cdots, n\}$ *such that* $\sum_{j \in J} a_j = B$.

Conversely, suppose that I_{ZSSC} has a solution; that is, there exists a simple cycle W such that $T(W) = \mathbf{0}$. Note that W must use e_1. Let $J = \{j \mid f_j \in W\}$. Then $\sum_{j \in J} a_j = B$. Thus I_{SS} has the solution J. □

6. Backedged two-terminal series-parallel multidigraphs. Two-Terminal Series-Parallel (TTSP) graphs have been well studied: the undirected version in [1], [8], [25], [27] because of its relationship to electrical networks and the directed version in [26] because it provides an algorithm to recognize general series-parallel digraphs.

A digraph is called a *multidigraph* if we allow multiple edges between the same two vertices. The definition of the class of TTSP multidigraphs appears in [26] as follows:

(1) A digraph consisting of two vertices joined by a single edge is in TTSP.

(2) If G_1 and G_2 are TTSP multidigraphs, so too is the multidigraph obtained by either of the following operations:

(a) *Two terminal parallel composition*: identify the source of G_1 with the source of G_2 and the sink of G_1 with the sink of G_2.

(b) *Two terminal series composition*: identify the sink of G_1 with the source of G_2.

Let TTSP (m) be the class of TTSP multidigraphs that have m edges.

From this definition, a TTSP multidigraph has a single source, denoted by s, and a single sink, denoted by t. Let G be a TTSP graph. A multidigraph, obtained by adding any number of *backedges* to a TTSP graph G, is called a BTTSP (*Backedged Two-Terminal Series-Parallel*) multidigraph. An edge (x, y) is called a *backedge* if there is a path from y to x in G. The graph G is called *the underlying* TTSP *graph of* G_B. Let BTTSP (m) be the class of BTTSP multidigraphs that have m edges. Fig. 5(a) shows an example of a BTTSP graph G_B that consists of a backedge indicated by dotted lines and the underlying TTSP graph G.

Let $G = (V, E, T)$ be a doubly weighted multidigraph with $V = \{v_1, v_2, \cdots, v_n\}$. Then for all v_i, v_j in V and $k \in \{1, 2, \cdots, n\}$, we define the convex polygon $\alpha_{ij}^k(T)$ in

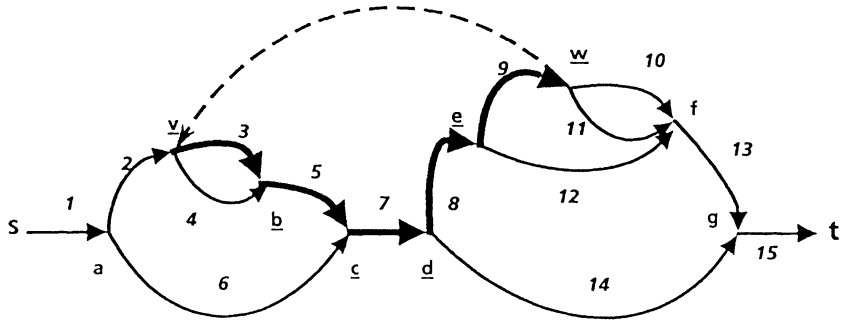


FIG. 5(a). *A* BTTSP *multidigraph* G_B; *a backedge is indicated by the dotted line.*

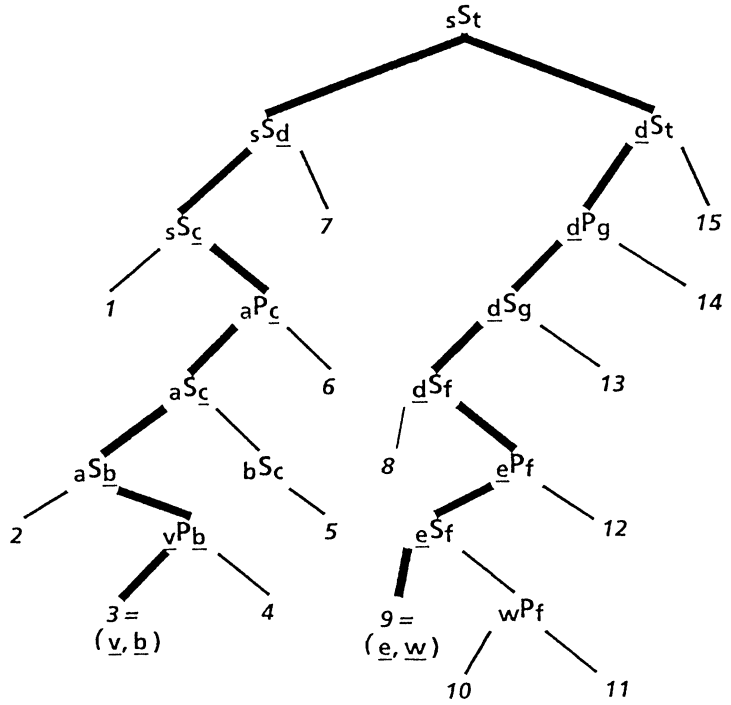


FIG. 5(b). *A binary decomposition tree* BDT(G). *The wide solid line corresponds to the path from v to w in Fig.* 5(a).

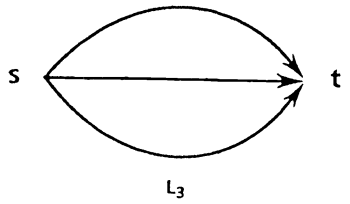


FIG. 5(c). $|\alpha_{st}(L_3)| \leqq 3$.

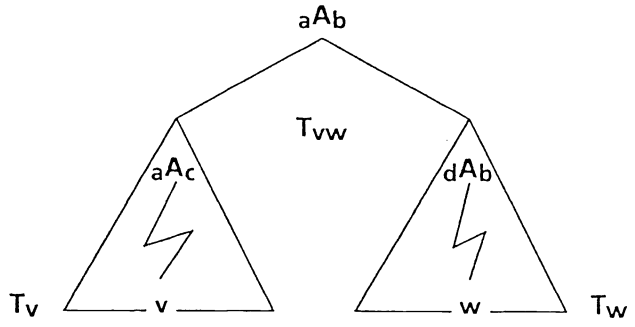


FIG. 5(d). ${}_aA_b = {}_aS_b$ *and* $c = d$. *Every path from* v *to* $x \notin T_v$ *passes through* c. *Every path from* $x \notin T_w$ *to* w *passes through* $d = c$.

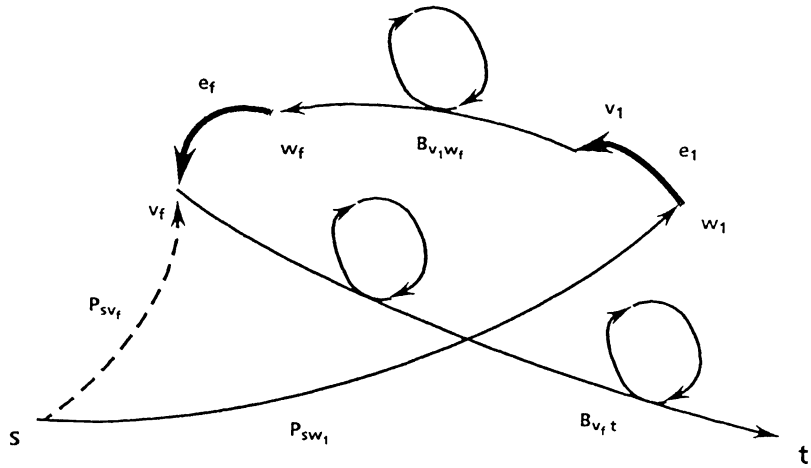


FIG. 5(e). *An edge* e_f *is the last backedge from which there is a path to* w_1. *Then we apply the induction hypothesis to the path* $P_{sv_f}B_{v_f t}$.

the same way as in the previous section: that is, $\alpha_{ij}^k(T)$ is the convex hull of all costs of paths in $PATH(v_i, v_j, k)$. In particular, we call $\alpha_{ij}^n(T)$ the *convex polygon* of $v_i - v_j$ *paths* and denote it by $\alpha_{ij}(T)$. For any multidigraph G, let $A(G) = \max_{i,j,k,T} |\alpha_{ij}^k(T)|$ and similarly for a class of graphs we write $A(\{G\})$. That is, $A(G)$ is the maximum number of edges in $\alpha_{ij}^k(T)$ when i, j, k, and T are arbitrary and G is fixed. We then have the following theorem.

THEOREM 6.1. *Let* G *be a doubly weighted multidigraph defined as above. For any* i, j, k, *and* T, *there exists a two-dimensional labeling* T' *such that* $\alpha_{ij}^k(T) = \alpha_{ij}(T')$. *Therefore,* $A(G) = \max_{i,j,T} |\alpha_{ij}(T)|$.

Proof. In order to prove the first part of the theorem, we only have to define $T'(e)$ as follows: (1) if e is on a path in $PATH(v_i, v_j, k)$, then define $T'(e) = T(e)$, and (2) otherwise define $T'(e) = \varnothing$. We then have $|\alpha_{ij}^k(T)| = |\alpha_{ij}(T')|$.

The second part of the theorem is immediate from the first part. □

From this result we can restrict attention to $\alpha_{ij}(T)$ instead of $\alpha_{ij}^k(T)$ in what follows. We now have the following theorem:

THEOREM 6.2. $A(\text{TTSP}(m)) = m$.

Before proving the theorem, we need some lemmas. Let L_m be the TTSP multidigraph consisting of two vertices s and t, and m edges from s to t. (See Fig. 5c.)

LEMMA 6.3. $A(L_m) = m$.

Proof. Let e_i for $i = 1, 2, \cdots, m$ be the edges of L_m, and let $T(e_i) = v_i \in Z \times Z$. Then α_{st} is the convex hull of $\{v_i\}$, which can clearly have m sides, and no more than m sides. □

LEMMA 6.4. *Let G be in* TTSP(m) *with source s and sink t, and let T be a two-dimensional labeling of G. Let x, y be arbitrary vertices in G such that $(x, y) \neq (s, t)$. Then there exists a two-dimensional labeling T' such that $|\alpha_{xy}(T)| \leqq |\alpha_{st}(T')|$.*

Before proving Lemma 6.4, we define the graph $G_{xy} = (V_{xy}, E_{xy})$ for $x, y \in V$ by the following operations on a TTSP graph $G = (V, E)$: (1) First, we delete all incoming edges to x and all outgoing edges from y. (2) We then delete all *useless* vertices and their adjacent edges. A vertex v is called *useless* when there is no $v - x$ path or $y - v$ path.

Proof of Lemma 6.4. If there is no $x - y$ path in G, we have $\alpha_{xy}(T) = \varnothing$. Thus $|\alpha_{xy}(T)| = 0 \leqq |\alpha_{st}(T)|$. Choose T as T'.

Otherwise there exists an $x - y$ path in G. Since there exists an $s - x$ path and a $y - t$ path, let $P_{sx}(P_{yt})$ be an arbitrary $s - x$ path ($y - t$ path). Let $G_1 = (V_1, E_1)$ be the graph consisting of P_{sx}, G_{xy}, and P_{yt}. We define a two-dimensional labeling T' as follows:

$$T'(e) = \begin{cases} \varnothing & \text{if } e \in E - E_1 \\ \mathbf{0} & \text{if } e \in P_{sx} \cup P_{yt} \\ T(e) & \text{if } e \in E_{xy}. \end{cases}$$

Then $|\alpha_{xy}(T)| = |\alpha_{st}(T')|$. □

We can now prove Theorem 6.2.

Proof of Theorem 6.2. We first prove $A(\text{TTSP}(m)) \leqq m$ by induction on m. It is clear that $A(\text{TTSP}(1)) = 1$. Assume that the induction hypothesis is true for $k < m$. Let $G = (V, E)$ be in TTSP(m) with source s and sink t. From Lemma 6.4, we only have to show $|\alpha_{st}(T)| \leqq m$ for any T. From the definition of TTSP, G must be constructed either in series or in parallel from $G_1 \in \text{TTSP}(m_1)$ and $G_2 \in \text{TTSP}(m_2)$ such that $m = m_1 + m_2$ and $m_1, m_2 > 0$. Then we have $A(G) \leqq A(G_1) + A(G_2) \leqq m_1 + m_2 = m$. Note that the first inequality uses Theorems 3.7 and 3.8, while the second uses the induction hypothesis. Thus $A(\text{TTSP}(m)) \leqq m$. Since $L_m \in \text{TTSP}(m)$, from Lemma 6.3, $A(L_m) = m$, which shows this bound is achievable. □

We will show the same result for the class of BTTSP multidigraphs. The following lemma says that every backedge in an $s - t$ path in a BTTSP graph lies on a cycle that lies on the $s - t$ path.

LEMMA 6.5. *Let G_B be a* BTTSP *graph with source s and sink t, and let P be a path from s to t possibly using some backedges in G_B. Then P can be represented as follows: $P = P_1 C_1^{r_1} P_2 C_2^{r_2} \cdots P_k C_k^{r_k}$ where $P_1 P_2 \cdots P_k$ is a path from source to sink in the underlying* TTSP *graph G, the C_i's are cycles in G_B, and $r_i \geqq 0$ for $1 \leqq i \leqq k$.*

Proof. For the proof see § 7. □

THEOREM 6.6. $A(\text{BTTSP}(m)) = m$.

Proof. Since TTSP$(m) \subset$ BTTSP(m), we have $m = A(\text{TTSP}(m)) \leqq A(\text{BTTSP}(m))$. We now prove that for an arbitrary graph $G_B \in \text{BTTSP}(m)$ with at least one backedge, $A(G_B) \leqq m$. Let $G = (V_1, E_1)$ be the underlying TTSP graph of G_B, and let T be a two-dimensional labeling of G_B. Let $P_B(s, t)$ be the set of $s - t$ paths in G_B, and let $P(s, t)$ be the set of $s - t$ paths in G. Let P be an arbitrary path in $P_B(s, t)$. Then from Lemma 6.5, P can be expressed as $P = P_1 C_1^{r_1} P_2 C_2^{r_2} \cdots P_k C_k^{r_k}$,

where $P_1P_2 \cdots P_k$ is a path from source to sink in the underlying TTSP graph G, the C_i's are cycles in G_B, and $r_i \geqq 0$ for $1 \leqq i \leqq k$. Let $\beta_P = T(P_1P_2 \cdots P_k)$ and $\gamma_{P_i} = T(C_i)$ for $1 \leqq i \leqq k$. Then $T(P) = \beta_P + \gamma_{P_1}^{r_1} + \gamma_{P_2}^{r_2} + \cdots + \gamma_{P_k}^{r_k}$. Let $P^* = \{P_1C_1^{n_1r_1}P_2C_2^{n_2r_2} \cdots P_kC_k^{n_kr_k} \mid P = P_1C_1^{r_1}P_2C_2^{r_2} \cdots P_kC_k^{r_k} \in P_B(s, t)$, and $n_i \in Z^+ \cup \{0\}$ for $1 \leqq i \leqq k\}$. Let $T(P^*)$ be defined as $T(P^*) = \uplus_{Q \in P^*} T(Q)$. Since $P^* \subset P_B(s, t)$, we have

$$T(P^*) = \beta_P + (\gamma_{P_1} \uplus \gamma_{P_2} \uplus \cdots \uplus \gamma_{P_k})^* \subset {}^{+}\bigcup_{P \in P_B(s,t)} T(P).$$

Note that $T(P) \subset T(P^*)$. Therefore $\uplus_{P \in P_B(s,t)} T(P) = \uplus_{P \in P_B(s,t)} T(P^*)$. Thus we now have

$$\begin{aligned}
|\alpha_{st}(T)| &= \left| \biguplus_{P \in P_B(s,t)} T(P) \right| = \left| \biguplus_{P \in P_B(s,t)} T(P^*) \right| \\
&= \left| \biguplus_{P \in P_B(s,t)} \beta_P + (\gamma_{P_1} \uplus \gamma_{P_2} \uplus \cdots \uplus \gamma_{P_k})^* \right| \\
&\leqq \left| \biguplus_{P \in P_B(s,t)} \beta_P \right| + 1 \quad \text{(using Theorem 3.11)} \\
&\leqq \left| \biguplus_{P \in P_B(s,t)} T(P) \right| + 1 \\
&\leqq |\alpha_{st}(G)| + 1 \quad \text{(from the definition)} \\
&\leqq A(\mathrm{TTSP}(|E_1|)) + 1 \\
&= |E_1| + 1 \quad \text{(using Theorem 6.2)} \\
&\leqq m
\end{aligned}$$

because $|E_1| \leqq m - 1$ by the assumption that G has a backedge. Thus $A(\mathrm{BTTSP}(m)) \leqq m$. □

COROLLARY 6.7. *For* BTTSP, *the algorithm* ZSC *runs in* $O(n^3m)$ *time where n is the number of vertices and m is the number of edges.*

Proof. The proof is clear from Theorems 4.4 and 6.6. □

7. Proof of Lemma 6.5. Let $G = (V, E)$ be a TTSP multidigraph with source s and sink t. A *binary decomposition tree for G*, denoted by BDT(G), which was discussed in [26], represents the construction process of G by a binary tree. A binary tree BDT(G) can be created by following the sequence of series and parallel compositions that construct G. Initially we have a set of singletons $\{e \mid e \in E\}$. Suppose we apply a two terminal parallel (respectively, series) composition to two TTSP graphs G_{xy} and G_{uv} and obtain the new TTSP graph G_{ab} where x, u, a are sources and y, v, b are sinks. Then we create BDT(G_{ab}) by creating the root ${}_aP_b$ (respectively, ${}_aS_b$) and make BDT(G_{xy}) a left subtree and BDT(G_{uv}) a right subtree. Thus in BDT(G), every leaf represents an edge in G and each internal node ${}_aP_b$ (respectively, ${}_aS_b$) represents a parallel (respectively, series) composition. Fig. 5(b) shows an example. Note that every path in G has a corresponding route in BDT(G). For example, the path

$$P = v - 3 - b - 5 - c - 7 - d - 8 - e - 9 - w,$$

shown in bold lines in Fig. 5(a), has the following corresponding route in BDT (G):

$$\begin{aligned} P_{\mathrm{BDT}}: \quad (\mathbf{v}, \mathbf{b}) &= 3 - {}_{v}P_{\mathbf{b}} - {}_{a}S_{\mathbf{b}} - {}_{a}S_{\mathbf{c}} - {}_{a}P_{\mathbf{c}} - {}_{s}S_{\mathbf{c}} - {}_{s}S_{\mathbf{d}} - {}_{s}S_{t} - {}_{\mathbf{d}}S_{t} \\ &\quad - {}_{\mathbf{d}}P_{g} - {}_{\mathbf{d}}S_{g} - {}_{\mathbf{d}}S_{f} - {}_{\mathbf{e}}P_{f} - {}_{\mathbf{e}}S_{f} - 9 \\ &= (\mathbf{e}, \mathbf{w}). \end{aligned}$$

Note that the vertices shown in bold face in the path P_{BDT}, (b, c, d, and e), appear in P in this order.

Let T_{vw} be the smallest subtree in BDT (G) that includes vertices v and w. (Find the nearest common ancestor and include the appropriate subtree.) Let T_v (respectively, T_w) be the subtree of T_{vw} in which v (respectively, w) exists as shown in Fig. 5(d). We use ${}_{a}A_{b}$ for representing either ${}_{a}S_{b}$ or ${}_{a}P_{b}$. Let ${}_{a}A_{b}$, ${}_{a}A_{c}$, and ${}_{d}A_{b}$ be the root of the subtrees T_{vw}, T_v, and T_w, respectively. Then we have the following lemma:

LEMMA 7.1. *Suppose ${}_{x}A_{y}$ appears in P_{BDT}. If ${}_{x}A_{y}$ appears in T_v, then y is in P. If ${}_{x}A_{y}$ appears in T_w, then x is in P.*

Proof. Suppose ${}_{x}A_{y}$ appears in T_v. The vertex v is in the TTSP graph with source x and sink y. Thus every path from v to a vertex that is not in T_v must pass through y. We can prove the other case in the same way. □

COROLLARY 7.2. *Suppose there is a $v-w$ path in G and T_v, T_w, and T_{vw} are defined as above. Let ${}_{a}A_{b}$, ${}_{a}A_{c}$, and ${}_{d}A_{b}$ be the roots of the subtrees T_{vw}, T_v, and T_w, respectively. Then we have the following*:

(1) *${}_{a}A_{b} = {}_{a}S_{b}$; that is, the root of T_{vw} corresponds to a series composition, and $c = d$.*

(2) *Every path from v to t passes through the vertex c.*

(3) *Every path from s to w passes through the vertex c.*

(4) *Any $v-t$ path and any $s-w$ path intersect at some vertex.*

Proof. (1) If the root of T_{vw} corresponds to a parallel composition, there is no path from v to w. Thus ${}_{a}A_{b} = {}_{a}S_{b}$. And the series composition identifies the sink of ${}_{a}C_{c}$ and the source of ${}_{d}A_{b}$, thus $c = d$.

(2) Since there is a $w-t$ path, $t \notin T_v$. Therefore, from the proof of the above lemma, every path from v to t passes through the vertex c.

(3) We can prove this in the same way as (2).

(4) This is obvious from (2) and (3). □

Proof of Lemma 6.5. Let k be the number of backedges in P. Let B_{xy} (P_{xy}) denote an $x-y$ path in G_B (G). We prove the lemma by induction on k.

Suppose $k = 1$ and let $e = (w, v)$ be the backedge in P. Note that there must be a $v-w$ path in the underlying TTSP graph G. P can be represented as $P = P_{sw}eP_{vt}$. P_{sw} and P_{vt} are paths in G, since $k = 1$, so that from Corollary 7.2, they pass through the same vertex c. Therefore, we can express P as $P = P_{sc}P_{cw}eP_{vc}P_{ct}$. Thus we obtain the cycle $C_1 = P_{cw}eP_{vc}$.

Suppose the lemma holds for numbers less than k. Let $E_B = \{e_1, e_2, \cdots, e_l\}$ be the backedges that appear in P in this order. Let $e_i = (w_i, v_i)$ for $1 \leqq i \leqq l$. Let $e_f = (w_f, v_f)$ be the last backedge in E_B such that there is a path from v_f to w_1 in G. Assume that $e_f \neq e_1$. (When $e_f = e_1$, we can easily modify the following proof.) Then as shown in Fig. 5(e), P can be represented as $P = P_{sw_1}e_1B_{v_1w_f}e_fB_{v_ft}$. Let P_{sv_f} be an arbitrary $s-v_f$ path in G, and let $P_1 = P_{sv_f}B_{v_ft}$. Then P_1 has l backedges where $l \leqq k-1$, because e_1 is not on P_1. From the induction hypothesis, P_1 can be formed from an $s-t$ path P_{st} and cycles $\{C_j \mid j \in J\}$. Note that P_{sv_f} is part of P_{st}; that is, there exists a v_f-t path P_{v_ft} such that $P_{st} = P_{sv_f}P_{v_ft}$. Suppose not. Let $e = (x, y)$ be the first edge in P_{sv_f} such that $y \notin P_{st}$. Then there exists a backedge (z, x) in B_{v_ft} and a cycle C_j such that $(z, x) \in C_j$. Since there exists an $x-v_f$ path and a v_f-w_1 path in G, there exists an $x-w_1$ path

in G. This contradicts the definition of v_f, since the backedge (z, x) is in $B_{v_f t}$, and thus appears after e_f in E_B.

Thus $P_{st} = P_{sv_f} P_{v_f t}$, and the path P consists of $P_{sw_1} e_1 B_{v_1 w_f} e_f$, $P_{v_f t}$, and cycles $\{C_j \mid j \in J\}$. Let $P_2 = P_{sw_1} e_1 B_{v_1 w_f} e_f P_{v_f t}$. Since there is a $v_f - w_1$ path in G, from Corollary 7.2, P_{sw_1} and $P_{v_f t}$ intersect at some vertex c. Thus P_2 can be expressed as $P_2 = P_{sc} \tilde{C} P_{ct}$ where $\tilde{C} = P_{cw_1} e_1 B_{v_1 w_f} e_f P_{v_f c}$. Therefore the path P consists of the path $P_{sc} P_{ct}$, cycle $\tilde{C}$, and cycles $\{C_j \mid j \in J\}$. □

8. Conclusion. We showed that the two operations of vector summation (+) and convex hull of union ($\uplus$) defined on the set of convex polygons form a closed semiring. We then investigated some properties of these operations. For example, the + operation can be done in $O(m)$ time, where m is the number of edges involved in the operation.

We then obtained the algorithm ZSC by using Kleene's closure algorithm on the above closed semiring. The algorithm ZSC solves the two-dimensional zero-sum cycle problem, which has a close relationship to the problem of acyclicity in two-dimensional regular electrical circuits. The complexities of our algorithm ZSC in some special cases are $O(n^3)$ time for the one-dimensional labeling case, $O(n^4 M)$ time for M-bounded graphs, and $O(n^3 m)$ time for BTTSP graphs, where n is the number of vertices and m is the number of edges. We also showed that the undirected version of the zero-sum cycle problem can be solved in $O(m \log m)$ time and that the zero-sum *simple* cycle problem is NP-complete.

We make the following conjecture about the number of edges of the convex polygons that appear in the algorithm ZSC:

CONJECTURE. *Let G, T, and $\alpha_{ij}(T)$ be defined in the same way as in the text. Then*

$$A(G) = \max_{i,j,T} |\alpha_{ij}(T)| \leqq m,$$

where m is the number of edges in G.

If this conjecture is true, then algorithm ZSC runs in $O(n^3 m)$ time on general graphs.

After the extended abstract of this paper appeared in [16], Kosaraju and Sullivan [18] showed that the zero-sum cycle problem for any dimension can be formulated in terms of linear programming, and thus is solvable in polynomial-time; Cohen and Megiddo [6] proved that the zero-sum cycle problem for any fixed dimension belongs to the class NC and can be solved in the two-dimensional case in serial time $O(nm)$. As mentioned in the Introduction, we hope the results in the present paper are of interest as a new connection between convex polygons and semirings, and as a novel application of Kleene's closure algorithm, even though faster algorithms are now available for the zero-sum cycle problem.

Acknowledgment. We are grateful to the referees for their careful reading of the paper and valuable comments that improved the quality of the paper. In particular, they suggested references [11], [20], [28], a simpler proof for Theorem 2.1 (3), and a better time complexity in Theorem 3.15.

REFERENCES

[1] A. ADAM, *On graphs in which two vertices are distinguished*, Acta Math. Acad. Sci., 12 (1961), pp. 377–397.

[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. CARRE, *Graphs and Networks*, Clarendon Press, Oxford, 1979.

[4] B. CHAZELLE AND D. DOBKIN, *Detection is easier than computation*, in Proc. 12th ACM Symposium on Theory of Computing, 1980, pp. 146-153.
[5] N. CHRISTOFIDES, *Graph Theory: An Algorithmic Approach*, Academic Press, London, 1975.
[6] E. COHEN AND N. MEGIDDO, *Strongly polynomial-time and* NC *algorithms for detecting cycles in dynamic graphs*, in Proc. 21th ACM Symposium on Theory of Computing, 1989, pp. 523-534.
[7] G. B. DANTZIG, W. O. BLATTNER, AND M. R. RAO, *Finding a cycle in a graph with minimum cost to time ratio with application to a ship routing problem*, in Internat. Symposium on Theory of Graphs, Dunod, Paris, 1967, P. Rosenstiehl, ed., Gordon and Breach, New York, pp. 77-83.
[8] R. J. DUFFIN, *Topology of series-parallel networks*, J. Math. Anal. Appl., 10 (1965), pp. 303-318.
[9] 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.
[10] M. GONDRAN AND M. MINGUX, *Graphs and Algorithms*, S. Vajda, trans., John Wiley, New York, 1984.
[11] B. GRÜNBAUM, *Convex Polytopes*, John Wiley, London, 1967.
[12] K. IWANO, *Two-dimensional dynamic graphs and their* VLSI *applications*, Ph.D. thesis, Computer Science Department, Princeton University, Princeton, NJ, 1987.
[13] K. IWANO AND K. STEIGLITZ, *Some experiments in* VLSI *leaf-cell optimization*, in VLSI Signal Processing, P. R. Cappello et al., eds., IEEE Press, New York, 1984, pp. 387-398; in Proc. IEEE 1984 Workshop on VLSI Signal Processing, Los Angeles, CA, November 27-29, 1984.
[14] ———, *Time-Power-Area tradeoffs for the* nMOS VLSI *full-adder*, in 1985 Proc. Internat. Conference on Acoustics, Speech, and Signal Processing, Tampa, FL, March 26-29, 1985, pp. 1453-1456.
[15] ———, *Optimization of one-bit full adders embedded in regular structures*, IEEE Trans. on Acoust., Speech Signal Process., ASSP-34 (1986), pp. 1289-1300.
[16] ———, *Testing for cycles in infinite graphs with periodic structure*, in Proc. 19th ACM Symposium on Theory of Computing, 1987, pp. 46-55.
[17] ———, *Planarity testing of doubly periodic infinite graphs*, Networks, 18 (1988), pp. 205-222.
[18] S. R. KOSARAJU AND G. F. SULLIVAN, *Detecting cycles in dynamic graphs in polynomial time*, in Proc. 20th ACM Symposium on Theory of Computing, 1988, pp. 398-406.
[19] E. L. LAWLER, *Optimal cycles in doubly weighted directed linear graphs*, in Internat. Symposium on Theory of Graphs, Dunod, Paris, 1967, P. Rosenstiehl, ed., Gordon and Breach, New York, pp. 209-213.
[20] L. A. LYUSTERNIK, *Convex Figures and Polyhedra*, T. J. Smith, trans., Dover, New York, 1963.
[21] K. MEHLHORN, *Data Structures and Algorithms* 2: *Graph Algorithms and* NP-*Completeness*, Springer-Verlag, Berlin, New York, 1984.
[22] J. B. ORLIN, *Some problems on dynamic/periodic graphs*, in Progress in Combinatorial Optimization, W. R. Pulleyblank, ed., Academic Press, Orlando, FL, 1984.
[23] C. H. PAPADIMITRIOU AND K. STEIGLITZ, *Combinatorial Optimization: Algorithms and Complexity*, Prentice-Hall, Englewood Cliffs, NJ, 1982.
[24] R. REITER, *Scheduling parallel computation*, J. Assoc. Comput. Mach., 15 (1968), pp. 590-599.
[25] J. RIORDAN AND C. E. SHANNON, *The number of two terminal series-parallel networks*, J. Math. Phys., 21 (1942), pp. 83-93.
[26] J. VALDES, R. E. TARJAN, AND E. L. LAWLER, *The recognition of series-parallel digraphs*, in Proc. 11th ACM symposium on Theory of Computing, 1-12, Atlanta, GA, April 1979.
[27] L. WEINBERG, *Linear graphs: Theorems, algorithms, and applications*, in Aspects of Network and System Theory, R. E. Kalman and N. DeClaris, eds., Holt, Rinehart, and Winston, New York, 1971.
[28] I. M. YAGLOM AND V. G. BOLTYANSKII, *Convex Figures*, P. J. Kelly and L. F. Walton, trans., Holt, Rinehart, and Winston, New York, 1961.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 902–906, October 1990

009

ON THE COMPLEXITY OF A GAME RELATED TO THE DICTIONARY PROBLEM*

K. MEHLHORN†, ST. NÄHER†, AND M. RAUCH†

Abstract. A game on trees that is related to the dictionary problem is considered. There are two players, A and B, which take turns. Player A models the user of the dictionary and player B models the implementation of it. At his turn, player A modifies the tree by adding new leaves and player B modifies the tree by replacing subtrees. The cost of an insertion is the depth of the new leaf, and the cost of an update is the size of the subtree replaced. The goal of player A is to maximize cost and the goal of B is to minimize it. It is shown that there is a strategy for player A, which forces a cost of $\Omega(n \log \log n)$ for an n-game, i.e., a game in which each player takes n turns, and that there is a strategy for player B, which keeps the cost within $O(n \log \log n)$.

Key words. dictionary problem, lower bound, hashing, search tree

AMS(MOS) subject classification. 68

1. Introduction.

We consider a two-person game on trees, which is related to the dictionary problem. The two players A and B take turns. Player A models the user of the dictionary and player B models the implementation of it. At his turn, player A modifies the tree by replacing a leaf by a tree consisting of a single node with two children, cf. Fig. 1. This is called an *insertion.* The *cost of an insertion* is the depth of the leaf replaced. At his turn, player B performs zero or more *updates.* An update replaces a subtree by another subtree with the same number of leaves. A precise definition is as follows.

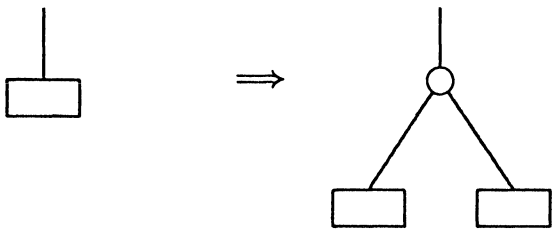


FIG. 1. *An insertion. The cost of an insertion is the depth of the new node v.*

Let T be a rooted tree and v a node of T. A *subtree* rooted at v is a connected subgraph of T with the following two properties:

(1) It contains v but no ancestor of v.

(2) If a node $w \neq v$ belongs to the subgraph, then all siblings of w also belong to the subgraph.

A subtree T' is *complete* if for every node w in T' all children of w also belong to the subtree. The complete subtree rooted at v is denoted by T_v.

* Received by the editors May 24, 1989; accepted for publication (in revised form) February 6, 1990. The research was partially supported by the Deutsche Forschungsgemeinschaft under grant SPP1.

† Fachbereich Informatik, Universität des Saarlandes, 6600 Saarbrücken, Federal Republic of Germany.

An *update* of a tree T is specified by a subtree T' of T rooted at some node v of T and a tree T'' having the same number of leaves as T'. The result of the update is a tree $\tilde{T}$, which can be obtained as follows:

(1) Delete all interior nodes of T' from T.

(2) Make the root of T'' a child of *parent*(v).

(3) Identify the ith leaf of T' with the ith leaf of T'' for $1 \leqq i \leqq m$, where m is the number of leaves of T'.

The *cost of an update* is the number of leaves of tree T'. Figure 2 gives an example for an update of cost 6. The trees T' and T'' are shown bold. The leaves of T' and T'' are numbered from 1 to 6.

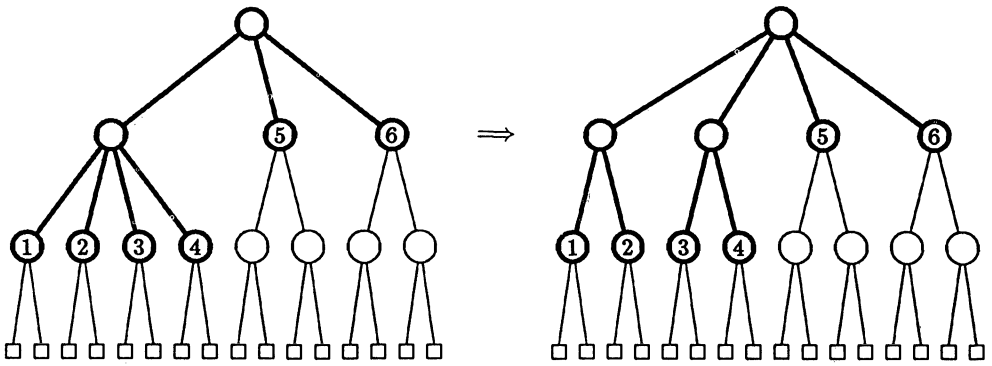


FIG. 2

An *n-game* starts with a trivial tree consisting of a single node, and ends after each player takes n turns. We will prove the following results:

THEOREM 1. *There is a strategy for player A such that the cost of any n-game is* $\Omega(n \log \log n)$.

THEOREM 2. *There is a strategy for player B such that the cost of any n-game is* $O(n \log \log n)$.

Theorems 1 and 2 are proven in §§ 2 and 3, respectively. Our game on trees models all solutions to the dictionary problem where a search is performed by repeated splitting of the dictionary until a dictionary of size one results. Special cases are multilevel hashing and search trees.

In (multilevel) hashing schemes a hash function is used for the splitting process, i.e., in each node of the tree a hash function h is stored, and the ith subtree of a node is a dictionary for all keys that are mapped to i by function h. Hashing with chaining as well as the perfect hashing schemes of Fredman, Komlós, and Szemeredi [FKS], Aho and Lee [AL], and Dietzfelbinger et al. [DKMMRT] are examples of multilevel hashing schemes. In these examples our definition of insertion cost fairly reflects the cost of an insertion, but our definition of update cost underestimates the true cost since it only measures the amount of structural change in the tree but ignores the cost of finding appropriate hash functions.

In search trees, e.g., AVL-trees or (2, 3)-trees, comparisons between keys are used for the splitting process. The updates correspond to small structural changes of the search tree; e.g., Fig. 2 shows how a node of degree 4 is split into 2 nodes of degree 2 in a (2, 3)-tree. In these examples our definition of update cost fairly reflects the true cost, but our definition of insertion cost underestimates the true cost because the

amount of work needed to identify the successor of a node on the search path depends on the degree of the node.

Together, these examples show that many solutions for the dictionary problem are within our model, and hence the lower bound given in Theorem 1 applies to them. The examples also indicate that the upper bound given in Theorem 2 does not imply a solution to the dictionary problem with the same performance, in fact, no such solution is known.

The work reported here extends work of Dietzfelbinger et al. [DKMMRT], who showed that player A can force a cost $\Omega(n \log n)$ in an n-game provided that player B always has to replace complete subtrees. This is a severe restriction and excludes balanced search trees. Note that Dietzfelbinger et al. [DKMMRT] have shown that in their model there is always a single turn (of either A or B), that has cost $\Omega(\sqrt{n})$ in any n-game. However, balanced search trees keep the cost of any turn in $O(\log n)$ and hence are excluded by their model. They are included in our model.

2. The lower bound. Player A follows a very simple strategy called the "insert-into-heaviest-child-strategy" (IHS). Let the weight $w(v)$ of a node v be the number of leaves of the complete subtree rooted at v. Then the path of insertion $v_0, v_1, \cdots, v_l$ is defined as follows: v_0 is the root of the current tree, v_{i+1} is a heaviest child of v_i, and v_l is a leaf.

THEOREM 1. *If player A plays according to the "insert-into-heaviest-child-strategy," then the cost of an n-game is* $\Omega(n \log \log n)$.

Proof. Consider any n-game where A plays IHS. We may assume without loss of generality that $n \geqq 4 \cdot 3^7$. Let $K = \max\{k \in \mathbb{N};\ k^{(2^k-1)} \leqq n/4\}$. Then $K = \Theta(\log \log n)$. We now distinguish cases. Assume first that there are at least $n/2$ insertions that have cost at least $K+1$ each. Then the total (insertion) cost is clearly $\Omega(n \log \log n)$.

Assume next that there are at least $n/2$ insertions with cost K or less. Let $D = (n/4)^{1/(2^K-1)}$. Then $D \geqq K \geqq 3$ by the definition of K. For a node v of tree T we use depth (v, T) to denote the depth of node v in T and deg (v, T) to denote the degree of node v in T; the depth of the root is 0. Nodes are created and destroyed by updates and insertions. Consider an update where subtree T' is replaced by subtree T''. Then T' and T'' have the same number of leaves and the update identifies the ith leaf of T' with the ith leaf of T''. We therefore say that the update *destroys* the interior nodes of T' and *creates* the interior nodes of T''. The leaves of T' are identified with the leaves of T'' and hence are said to *exist* before and after the update. In the example of Fig. 2 the update destroys two nodes and creates three nodes. An insertion creates a leaf and a node of degree 2. With this definition, the degree of a node never changes during its existence. We may therefore write deg (v) instead of deg (v, T). The depth of a node v, however, can change over time. For the ith update we use V_i to denote the set of nodes of degree 2 or more created by the update. Also, $V = \bigcup_{i=1}^{n} V_i$ is the set of nodes that are created by updates and have degree two or more.

LEMMA 1. *Consider an update of cost C. Let V' be the set of nodes of degree at least 2 that are created by the update. Then* $2C \geqq \sum_{v \in V'} \deg(v)$.

Proof. This follows from the simple observation that $\sum_{v \in V'} \deg(v)$ is at most twice the number of leaves of the new subtree T''. □

We call a node v of T *big* (*with respect to* T) if depth $(v, T) < K$ and deg $(v) \geqq D^{2^{K-\text{depth}(v,T)-1}}$. Note that an update may change the status of a node from big to nonbig and vice versa by changing the depth of the node. Since $D \geqq 3$, only nodes created by updates can ever be big.

LEMMA 2. *Every insertion of cost at most K, except the first $n/4$, goes through a big node, i.e., the path of insertion contains a node which is big with respect to the current tree.*

Proof. Let $v_0, v_1, \cdots, v_l$ with $l \leqq K$ be the path of insertion. Let d_j be the degree of node v_j and let w_j be the weight of node v_j, $0 \leqq j \leqq l$. Then $w_l = 1$, since v_l is a leaf, $w_{l-1} \leqq d_{l-1} \cdot w_l$ since A plays IHS, and $w_0 \geqq n/4$. Thus $n/4 \leqq \prod_{j=0}^{l-1} d_j$ and hence $d_j \geqq D^{2^{K-j-1}}$ for some j. □

Since there are $n/2$ insertions of cost K or less, we conclude that there are $n/4$ insertions that go through a big node. For every such insertion I let $v(I)$ be the big node of largest depth on the path of insertion; we say that I is *assigned* to v. For every node v let $n(v)$ be the number of insertions assigned to v. Note that $n(v) > 0$ implies $v \in V$ and $\sum_{v \in V} n(v) \geqq n/4$.

LEMMA 3. $n(v) \leqq \deg(v)/D$ *for every vertex* $v \in V$.

Proof. Consider any vertex v. Let i be the minimal integer such that $\deg(v) \geqq D^{2^{K-i-1}}$. Consider any insertion I that is assigned to v. As in the proof of Lemma 2, let $v_0, v_1, \cdots, v_l$ with $l \leqq K$ be the path of insertion, let d_j be the degree of node v_j, and let w_j be the weight of node v_j, $1 \leqq j \leqq l$. Since $v = v(I)$ we have $v = v_{i+p}$ for some $p \geqq 0$. Also, the nodes $v_{i+p+1}, v_{i+p+2}, \cdots, v_{l-1}$ are not big at depths $i+p+1, \cdots, l-1$, respectively, and hence $d_{i+p+1} \leqq D^{2^{K-(i+p+1)-1}}, \cdots, d_{l-1} \leqq D^{2^{K-(l-1)-1}}$. Thus $w_{i+p+1} \leqq D^{2^{K-i-2}} \cdot D^{2^{K-i-3}} \cdots D^{2^0} \leqq D^{2^{K-i-1}-1} \leqq \deg(v)/D$. □

It is now easy to complete the proof of Theorem 1. We have by Lemmas 1–3

$$\begin{aligned} \textit{total update cost} &\geqq \sum_{i=1}^{n} \sum_{v \in V_i} \deg(v)/2 \\ &\geqq \sum_{v \in V} n(v) \cdot D/2 \\ &\geqq n \cdot D/8 \\ &= \Omega(n \log\log n). \end{aligned}$$

□

3. The upper bound. Player B maintains a (balanced) tree where

(1) all leaves have the same depth, and

(2) the nodes of height i (leaves have height 0) have degree at most $2 \cdot 2^{2^{(i-1)}}$ and at least degree $2^{2^{(i-1)}}$, except for the root which has degree at least 2.

Clearly, if such a tree has depth K then it has at least $2^{2^{(K-2)}}$ leaves and hence any insertion in an n-game has cost $O(\log\log n)$.

Player B updates the tree as follows. He first restores property (1) by incorporating the new subtree into its parent (see Fig. 3).

This has cost at most 5 and may create a node of degree 5. Player B then walks back to the root. When it encounters a node of degree $2 \cdot 2^{2^{(i-1)}} + 1$ at height i, it then splits the node into two nodes of degree $2^{2^{i-1}}$ and $2^{2^{i-1}} + 1$, respectively, and increases the degree of the parent node. This has cost $O(2^{2^i})$. If the root is split a new root of degree 2 is created. In this way property (2) is maintained.

We next compute the total update cost. Let s_i be the number of times a node of height i is split. Then $s_1 \leqq n$, clearly, and $s_i \leqq s_{i-1}/2^{2^{(i-1)}}$, since a split at height $i-1$ increases the degree of a node at height i by one, nodes at height i start with degree at most $2^{2^{i-1}} + 1$, and split when their degree reaches $2 \cdot 2^{2^{i-1}} + 1$. Thus $s_i \leqq n/2^{2^{i-2}}$. The total update cost is

$$O\left(n + \sum_{i \leqq O(\log\log n)} s_i \cdot 2^{2^i}\right) = O(n \log\log n).$$

This proves Theorem 2.

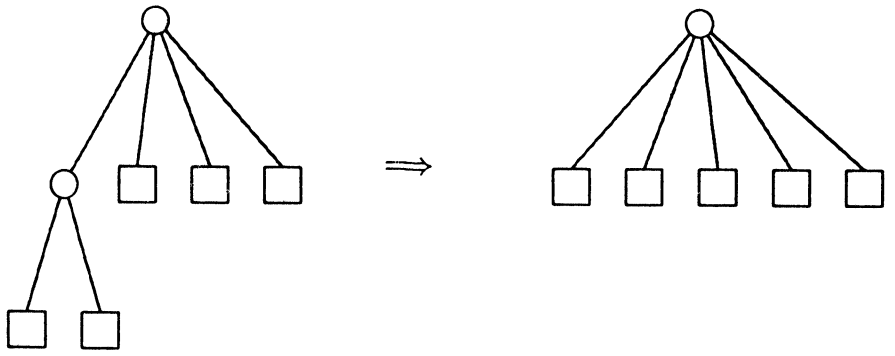


FIG. 3. *B's first move after an insertion.*

4. Conclusion. We studied the complexity of a game on trees. The game encompasses many solutions to the dictionary problem, in particular all balanced tree and multilevel hashing schemes. We have shown that in our model the amortized cost of maintaining a dictionary of size n is $\Theta(\log \log n)$.

REFERENCES

[AL] H. V. AHO AND D. LEE, *Storing a dynamic sparse table*, 27th IEEE Symposium on Foundations of Computer Science, 1986, pp. 55-60.

[DKMMRT] M. DIETZFELBINGER, A. KARLIN, K. MEHLHORN, F. MEYER AUF DER HEIDE, H. ROHNERT, AND R. TARJAN, *Upper and lower bounds for the dictionary problem*, in Proc. 29th IEEE Symposium on Foundations of Computer Science, 1988.

[FKS] M. L. FREDMAN, J. KOMLÓS, AND E. SZEMERÉDI, *Storing a sparse table with $O(1)$ worst case access time*, J. Assoc. Comput. Mach., 31 (1984), pp. 538-544.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 907-911, October 1990

010

LINEAR-TIME TEST FOR SMALL FACE COVERS IN ANY FIXED SURFACE*

D. BIENSTOCK†

Abstract. For any fixed surface S and fixed integer $k \geqq 0$, a linear-time algorithm is presented that tests whether selected vertices of a graph drawn on S can be covered with k or fewer faces.

Key words. graph algorithms, polynomial time

AMS(MOS) subject classifications. 05C10, 68

1. Introduction. Consider the following computational problem, called *FACE COVER* or *DISK DIMENSION* (refer to [BM], [FL], [FLS], or [J] for background). Given a graph G, drawn on a surface S, a subset $X \subseteq V(G)$, and an integer $k \geqq 0$, can the elements of X be covered with at most k faces? This problem has two variants:

(i) G is given with a fixed embedding in S,

(ii) We must optimize over all possible embeddings of G in S.

This problem has applications in various algorithmic settings. In particular, we mention the recent work of Schrijver on homotopic routings, where a polynomial-time algorithm for (i) is needed (see [S] for the planar graph case, later extended to arbitrary surfaces).

In [BM] it was shown that if S is the sphere (the planar case) then (i) and (ii) are NP-complete, but that for each fixed k both (i) and (ii) can be solved in linear time. More precisely, the complexity of the algorithm is at most $2^{ck}n$, where n is the number of vertices and c is a fixed constant.

The purpose of this paper is to show that for *any* fixed surface S and fixed $k \geqq 0$, there is a linear-time algorithm for (i). The algorithm described here will use the one given in [BM] as a subroutine, but otherwise this paper is self-contained: any linear-time algorithm for the planar case of (i) could be used instead. We remark that, when dealing with a problem in topological graph theory, it is often desirable to reduce the problem on an arbitrary surface to a similar problem in a simpler surface, in particular to a similar problem concerning planar graphs. This reduction is not always an easy task. This paper shows that, for the face cover problem, the reduction can indeed be easily carried out, and in fact the complexity of the overall algorithm remains linear in the size of the original problem.

We will need some facts and definitions. We refer the reader to [M] for topological background. Some of our definitions are similar to those in [RS1]. A *surface* is a compact connected two-dimensional manifold with boundary. We denote by $\Sigma(a, b, c)$ the surface obtained by adding to a sphere a handles and b crosscaps, and cutting out c pairwise disjoint open disks. Every surface is homeomorphic to some $\Sigma(a, b, c)$.

If S is homeomorphic to $\Sigma(a, b, c)$, the *Euler characteristic* of S, $\chi(S)$, equals $2-(2a+b+c)$, and its *number of boundary components*, $\gamma(S)$, is c. Given S, an *O-arc* in S is a homeomorphic image of the unit circle $\{(x, y) \in \mathbf{R}^2: x^2+y^2=1\}$. An *I-arc* is

* Received by the editors February 3, 1989; accepted for publication (in revised form) February 13, 1990. This research was carried out while the author was at Bell Communications Research, Morristown, New Jersey 07960.

† Department of Industrial Engineering and Operational Research, Columbia University, New York, New York 10027.

a homeomorphic image of the unit interval $\{0 \leqq x \leqq 1\}$; the images of 0 and 1 are the *ends* of the I-arc.

For convenience, we will use the same terminology to refer to a graph and a given drawing of it on a surface. Thus if G is drawn on S, $V(G)$ corresponds to a finite subset of S, each edge is an I-arc with corresponding ends, and different edges may meet only at their ends. We will assume that any drawing of a graph is such that all intersections with the boundary of the surface occur at vertices. The subset of a surface given by the drawing of a graph G is denoted by $D(G)$. If $D(G)$ is removed from the surface, the resulting pieces of the surface are called *faces*. The *dual* graph of G is the (combinatorial) graph whose vertices are the faces of G, and such that two faces are adjacent if they are incident to a common edge of G.

Given G, drawn in S, and O- or I-arc Γ is called *G-standard* if

(i) $\Gamma \cap V(G) \neq \emptyset$,

(ii) $\Gamma \cap D(G)$ is finite and each intersection point is a crossing, and

(iii) Γ crosses each face of G at most once.

2. The algorithm. Before describing the algorithm we need some observations. Let us first show how to extend a linear-time algorithm for the planar case of problem (i) to handle graphs with *forbidden faces* (that is, faces that cannot be used for covering). (We remark that this feature is easily built into the algorithm of [BM] without worsening the linear-time performance.)

Suppose we want to forbid a face f. Then subdivide each edge in the boundary of f once, add a new vertex drawn in f, and join this vertex to all vertices in the closure of f (including the new ones). It is seen that, without loss of generality, none of the new faces will be used towards a cover.

The second observation is that we can transform our covering problem to one of covering distinguished *faces*. As above, let $X \subseteq V(G)$ be the set of vertices to be covered. Then we expand each $x \in X$ into a small polygon $p(x)$, such that each vertex of $p(x)$ is adjacent to a distinct neighbor of x, with $p(x)$ drawn in S in the obvious way. We denote by G_0 the resulting graph and by X_0 the set of polygons resulting from X. The problem instance is denoted by (G_0, X_0). We write $\tau(G_0, X_0)$ for the covering number, i.e., the minimum cardinality of a set of faces Z of G_0, none of them in X_0, such that every element of X_0 is adjacent to an element of Z in the dual graph G_0^* of G_0.

We need some preliminary results.

(2.1) We may assume that every face of G_0 is in X_0 or is adjacent to an element of X_0.

Proof. If not, we can remove an edge and obtain an equivalent problem. □

In what follows we will assume (2.1) holds. The following result is found in [BM], for the planar case, and its proof is not different here, but we include it for completeness.

(2.2) If $\tau(G_0, X_0) \leqq k$ then the diameter of G_0^* is at most $8k+7$.

Proof. Let Z be a cover of X_0, $|Z| \leqq k$. Then the maximum distance in (G_0^*) from any face f to Z is at most one if f is in X_0, and at most two, otherwise, by (2.1). The conclusion follows by considering every fourth face in a diameter of G_0^*. □

Remark 1. The computation of the precise diameter of G_0^* may require more than linear time. However, the main implication of (2.2) is that the diameter may be assumed to be at most $O(k)$. This can be tested in linear time by computing the distance from a fixed face to all others. If true, then we can construct a path of length at most $16k+14$ between any two faces.

An O-arc Γ is called *critical* if

(i) Γ is G_0-standard and disjoint from the boundary of S.

(ii) Γ is non-null-homotopic in the surface obtained from S by pasting a disk onto each boundary component.

Remark 2. A critical O-arc Γ may be *orientation reversing*, that is traveling once around Γ will "reverse" the local orientation on S. In this case a narrow band of S cut along Γ will be homeomorphic to a Möbius strip. If Γ is not orientation reversing, then it essentially loops around a handle of S.

A G_0-standard O-arc is called *short* if it intersects at most $32k+29$ faces.

The main result that drives our algorithm is the following.

(2.3) Suppose $2-\chi(S)-\gamma(S)>0$ and $\tau(G_0, X_0)\leqq k$.
Then S contains a short critical O-arc.

Proof. Since $2-\chi(S)-\gamma(S)>0$, there is a critical O-arc Γ_1. Suppose Γ_1 is not short; let f_1, f_2 be two faces intersected by Γ_1, at distance $16k+15$ along Γ_1. Let p_i, $i=1,2$, be points in f_i contained in Γ_1. Then Γ_1 is the union of two I-arcs J, J', with ends p_1, p_2 (say J intersects $16k+16$ faces). There is a path in G_0^* between f_1 and f_2 of length at most $16k+14$ (see Remark 1); this gives rise to a G_0-standard I-arc L with ends p_1, p_2 which intersects at most $16k+15$ faces. Then at least one of $J\cup L$, $J'\cup L$ includes a critical O-arc. In the first case we obtain a short critical O-arc; in the second, a new critical O-arc that intersects fewer faces than Γ_1 is obtained. Proceeding inductively, we obtain the desired result. □

Using (2.3), we can now obtain a linear-time algorithm.

Step 1. Reduction to the planar case. Let Γ be a short critical O-arc, and let S_1 be the surface obtained from S by cutting along Γ. It may be verified that $\chi(S_1)=\chi(S)$, but that S_1 has one more boundary component than S. Thus $2-\chi(S_1)-\gamma(S_1)<2-\chi(S)-\gamma(S)$; in other words, S_1 is a simpler surface than S. Each intersection point of Γ_1 with $D(G_0)$ gives rise to two points located in the additional boundary component of S_1; we draw new vertices at all such points of S_1 to obtain a graph G_1 on S_1. Since Γ is G_0-standard, cutting along Γ splits some faces of G_0 into *pairs* of faces of G_1. We call these faces of G_0 *split faces*. Each face of G_0 that is not split gives rise to a unique face of G_1. Let X_1 be the set of faces of G_1 resulting from X_0 (split or otherwise).

It is not difficult to see that if it is the case that $\tau(G_0, X_0)\leqq k$, then one also has $\tau(G_1, X_1)\leqq O(k)$. Thus we can repeat the same basic procedure: test for short distance between all faces of G_1, and if verified, cut S_1 along a (properly defined) short critical O-arc. Proceeding inductively, we will either at some point conclude that $\tau(G_0, X_0)>k$, or after $m-1=2-\chi(S)-\gamma(S)$ steps, we will arrive at a triple S_{m-1}, G_{m-1}, X_{m-1}, where $2-\chi(S_{m-1})-\gamma(S_{m-1})=0$. In other words, S_{m-1} is homeomorphic to a sphere with $\gamma(S_{m-1})$ holes.

Next, we transform S_{m-1} into a surface S_m homeomorphic to a sphere by pasting a disk onto each boundary component. Also, we transform G_{m-1} into a graph G_m by joining any two consecutive vertices along a boundary component with an edge, drawn as the corresponding segment of boundary component. Thus G_m contains some new special faces that we call *boundary faces*, each corresponding to a hole in S_{m-1}, and there is a bijection between the remaining faces of G_m and the faces of G_{m-1}. Let X_m correspond to X_{m-1}.

Step 2. Computing $\tau(G_m, X_m)$. For the final step, we would like to use any linear-time algorithm for the face cover problem in the plane, which tests for small

$\tau(G_m, X_m)$, with the boundary faces forbidden. But before we can do this, we have to take care of the fact that some faces of G_0 have been split. If a face f of G_0, not in X_0, was split, then we must use either none or all of its "descendants" in G_m towards a cover of X_m. Furthermore, in the latter case, we must charge the use of all the descendants (to the value of the face cover) as costing only one unit. Finally, if $f \in X_0$ is split, then we only have to cover *at least one* descendant of f, not necessarily *all of them.*

This task is simple, however, because the total number of split faces of G_0 is small: at most $O(k \cdot 2^{(2-\chi(S)-\gamma(S))})$, because all the cuts are short. The total number of descendants of any split face in X_0 is also small, at most $2^{(2-\chi(S)-\gamma(S))}$. This allows us to try all possible choices of descendants. Let J_0 denote the set of split faces of G_0 that are not in X_0.

In short, for each split face in X_0 we choose a "representative" descendant in X_m, which defines a set $X_m^1 \subseteq X_m$ to be covered. We also select a subset $F_0 \subseteq J_0$, all of whose members are to be forbidden towards a cover (their descendants in G_m will all be forbidden). The remaining set A_0 of faces in J_0 are all to be used; we remove from X_m^1 any faces covered by descendants of elements of A_0. This defines a set $X_m^2 \subseteq X_m^1$.

Our task is then to check whether $\tau(G_m, X_m^2) \leqq k - |A_0|$, where all the descendants of faces in F_0, as well as the boundary faces, are forbidden. For this test we can now use any linear-time algorithm for testing small face covers in the plane, extended to handle forbidden faces as described before. This concludes the description of the algorithm.

To analyze the complexity of the overall algorithm, we note that the total number of choices in Step 2 is bounded, it is at most $O(k \cdot 2^{(2-\chi(S)-\gamma(S))})$. Hence the algorithm runs in time $O(n)$, as desired, pending some simple technical issues concerning the algorithmic handling of surfaces, which are described below. We point out that if we use the algorithm of [BM] as the subroutine for solving planar face cover problems, then the constant in the $O(\ )$ will be singly exponential in k and in $2-\chi(S)-\gamma(S)$. Whether this can be improved is an open problem.

In the discussion above we did not mention how the surface S is to be represented, and how the drawing of G on S is given, because these issues are not really relevant to the algorithm, and it is easy to pass from one representation to the other. It is particularly simple to use a *polygon representation* of S. Here S is given by a polygon P, some of whose sides are to be identified in pairs, with prescribed orientations. Any surface can be obtained in this manner (see [M]). A graph drawn in S has essentially a planar representation in P. The polygon representation makes it an easy task to find a critical O-arc in linear time, and given such an O-arc, (2.3) yields a linear-time procedure that produces a short critical O-arc. Other tasks, such as constructing dual graphs, are also easily seen to require linear time.

3. Concluding remarks. It would be interesting to obtain linear-time algorithms for the nonfixed embedding version (ii) of the face cover problem (it is known that there are polynomial-time algorithms for this problem, see [RS2]). One reason that makes this task difficult is the following. For planar graphs there is a structure theorem that explains when a graph has a unique plane embedding (precisely when the graph is 3-connected), and how to generate all embeddings, otherwise. This result is used explicitly in the linear-time algorithm of [BM] for version (ii), as well as used implicitly in most planarity testing algorithms. However, for higher surfaces (even simple ones)

the known theorems are not as strong. See [N] for the projective plane case. This remains an interesting open area of research.

It should be added that the Robertson-Seymour results on graph minors yield that there exist $O(n^3)$ algorithms for (ii). Unfortunately, these results are nonconstructive in the sense that one knows that such algorithms exist, without explicitly having the algorithms. Recently, Fellows and Langston have been able to improve some of the results arising from the Robertson-Seymour constructions, and for case (ii) the improvements may yield an explicit $O(n^3 \log n)$ algorithm (personal communication).

The face cover problem is also interesting from a nonalgorithmic point of view. Notice that the face cover problem is a special case of the set cover problem. Consequently, the related packing problem should be of interest. In the language of this paper, the packing problem arises as follows. Given an instance of the face cover problem with graph G and distinguished subset X, an obvious lower bound to $\tau(G, X)$ is the maximum cardinality of a subset of X, with the property that no two elements of the subset lie on a common face of G. Let us denote this parameter $\nu(G, X)$. What is the relationship between $\tau(G, X)$ and $\nu(G, X)$? In [BD] it is shown that for any fixed surface S there is a constant $c = c(S)$, so that for any G and X, $\tau(G, X) \leqq c\nu(G, X)$. In other words, $\tau(G, X)$ and $\nu(G, X)$ are always of the same order of magnitude. Therefore, in particular, there is a polynomial-time approximation algorithm for computing $\tau(G, X)$, based on linear programming, with bounded ratio in any fixed surface. We stress here that these results apply to *any* G and X, not just the "fixed k" cases.

REFERENCES

[BD] D. BIENSTOCK AND N. DEAN, *On obstructions to small face covers in embedded graphs*, submitted.

[BM] D. BIENSTOCK AND C. L. MONMA, *On the complexity of covering vertices by faces in a planar graph*, SIAM J. Comput., 17 (1988), pp. 53-76.

[FL] M. R. FELLOWS AND M. A. LANGSTON, *Nonconstructive tools for proving polynomial-time decidability*, J. Assoc. Comput. Mach., 35 (1988), pp. 727-739.

[FLS] M. R. FELLOWS, M. A. LANGSTON, AND M. SYSLO, *A topological parameterization and hard graph problems*, Congr. Numer., 59 (1987), pp. 69-78.

[J] D. S. JOHNSON, *The* NP*-completeness column: An ongoing guide*, J. Algorithms, 8 (1987), pp. 285-303.

[M] W. S. MASSEY, *Algebraic Topology: An Introduction*, Springer-Verlag, Berlin, New York, 1967.

[N] S. NEGAMI, *Re-embedding of projective-planar graphs*, J. Combin. Theory Ser. B, 44 (1988), pp. 276-299.

[RS1] N. ROBERTSON AND P. D. SEYMOUR, *Graph Minors. VII. Disjoint paths on a surface*, J. Combin. Theory Ser. B, 45 (1988), pp. 212-254.

[RS2] ———, *Disjoint paths—A survey*, SIAM J. Algebraic Discrete Methods, 6 (1985), pp. 300-305.

[S] A. SCHRIJVER, *Disjoint homotopic paths and trees in planar graphs*, manuscript, 1989.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 912–924, October 1990

011

OPTIMAL SIZE INTEGER DIVISION CIRCUITS*

JOHN H. REIF† AND STEPHEN R. TATE†

Abstract. Division is a fundamental problem for arithmetic and algebraic computation. This paper describes Boolean circuits (of bounded fan-in) for integer division (finding reciprocals) that have size $O(M(n))$ and depth $O(\log n \log\log n)$, where $M(n)$ is the size complexity of $O(\log n)$ depth integer multiplication circuits. Currently, $M(n)$ is known to be $O(n \log n \log\log n)$, but any improvement in this bound that preserves circuit depth will be reflected by a similar improvement in the size complexity of our division algorithm. Previously, no one has been able to derive a division circuit with size $O(n \log^c n)$ for any c, and simultaneous depth less than $\Omega(\log^2 n)$. The circuit families described in this paper are logspace uniform; that is, they can be constructed by a deterministic Turing machine in space $O(\log n)$.

The results match the best-known depth bounds for logspace uniform circuits, and are optimal in size.

The general method of high-order iterative formulas is of independent interest as a way of efficiently using parallel processors to solve algebraic problems. In particular, this algorithm implies that any rational function can be evaluated in these complexity bounds.

As an introduction to high-order iterative methods a circuit is first presented for finding polynomial reciprocals (where the coefficients come from an arbitrary ring, and ring operations are unit cost in the circuit) in size $O(PM(n))$ and depth $O(\log n \log\log n)$, where $PM(n)$ is the size complexity of optimal depth polynomial multiplication.

Key words. algebraic computation, integer division, circuit complexity, powering

AMS(MOS) subject classifications. 68Q25, 68Q40

1. Introduction.

In arithmetic and algebraic computation, the basic operations are addition, subtraction, multiplication, and division. It is a fundamental problem to find efficient algorithms for division, as it seems to be the most difficult of these basic operations. Problems are studied with both sequential models (Turing machines or bit-operation RAMs) and parallel models (circuits and bit-operation PRAMs); the model that we use in this paper is the circuit. A circuit is an acyclic directed graph with a set of nodes designated as input nodes (with zero fan-in), a set of nodes designated as output nodes (with zero fan-out), and a function basis with the elements labeling all noninput nodes. The value at any node is computed by applying the function labeling that node to the values of its predecessors, which are found in the same way — this goes on recursively until the input nodes are reached. Assigning a vector of values to the input nodes and computing the value of each output node, a circuit can be viewed as computing a function over vectors in the value domain. All circuits discussed in this paper have the additional restriction that every node must have fan-in bounded by some constant (without loss of generality, we can assume that every node has no more than two predecessors). The *size* of a circuit is the number of nodes in the circuit, and the *depth* of the circuit is the length of the longest path from an input node to an output node. The circuits used in most of this paper have function basis made up of the Boolean functions AND, OR, and NOT; these are called bounded fan-in Boolean circuits and are the standard model for arithmetic

* Received by the editors September 28, 1988; accepted for publication (in revised form) January 24, 1990. This research was supported by National Science Foundation grant CCR-8696134, by grants from the Office of Naval Research under contracts ONR-N00014-87-K-0310 and ONR-N00014-88-K-0458, by the Defense Advanced Research Projects Agency contract DAAL03-88-K-0195, and by the Air Force Office of Scientific Research contract AFOSR-87-0386.

† Department of Computer Science, Duke University, Durham, North Carolina 27706.

computation. In § 3 (dealing with polynomial reciprocals) we use a circuit model with operations in an arbitrary ring as the basis.

Optimal algorithms have been known for quite some time for addition and subtraction, and good algorithms exist for multiplication. If we let $SM(n)$ be the sequential time complexity of multiplication and $M(n)$ be the size complexity of $O(\log n)$ depth multiplication using the circuit model, then the best known results are due to Schönhage and Strassen [11] who give an algorithm based on discrete Fourier transforms with $SM(n) = O(n \log n \log\log n)$ and $M(n) = O(n \log n \log\log n)$.

The problem of integer division was examined by Cook in his Ph.D. thesis [5], and it was shown by using second-order Newton approximations that the sequential time complexity of taking reciprocals is asymptotically the same as that of multiplication. Unfortunately, this method does not carry over to the circuit model — for size $O(M(n))$ division circuits, we require depth $\Omega(\log^2 n)$ from a direct translation of Cook's method of Newton iteration. In addition, no one has been able to derive a new method for integer division with size $O(M(n))$ and depth less than $\Omega(\log^2 n)$ until now.

A long-standing open question has been to match the optimal depth bounds obtained for addition, subtraction, and multiplication with a division circuit of polynomial size. Until 1983, no one had presented a circuit for finding reciprocals with polynomial size and depth better than $\Omega(\log^2 n)$, then Reif presented a logspace uniform circuit based on wrapped convolutions with depth $O(\log n(\log\log n)^2)$ and slightly more than polynomial size [8]. A year later Beame, Cook, and Hoover presented a polynomial time uniform circuit based on Chinese remaindering with polynomial size and depth $O(\log n)$ [3]. A revised paper by Reif reduced the depth bounds on the logspace uniform circuit to $O(\log n \log\log n)$ while simultaneously achieving polynomial size [9]. For giving deterministic space bounds, logspace uniform circuits are vital as explained by Borodin [4]; in addition, the polynomial time uniform circuits that have been given use polynomial size tables of precomputed values, which a purist might find objectionable.

The size bounds for the above circuits are at least quadratic, and further work has been done to decrease the size bounds while keeping the depth the same. Shankar and Ramachandran [12] make a significant step in this direction by using discrete Fourier transforms to reduce the problems in size. They then apply either Reif's circuit (to give a logspace uniform circuit), or the Beame, Cook, and Hoover circuit (to give a polynomial time uniform circuit). The best depth bounds for each type of circuit are matched, and the size of both circuits is $O(n^{1+\epsilon}/\epsilon^4)$, for any sufficiently small $\epsilon > 0$. Independent work on a polynomial time uniform circuit by Hastad and Leighton [6] resulted in an efficient circuit for Chinese remaindering which gave a division circuit of size $O(n^{1+\epsilon})$ and depth $O((1/\epsilon^2)\log n)$, for $\epsilon > 0$.

Until 1988, no one had given a circuit with depth less than $\Omega(\log^2 n)$, and simultaneous size $O(n \log^c n)$ for any c. A preliminary version of this paper [10] gave logspace uniform circuits that have size $O(M(n))$ and depth $O(\log n(\log\log n)^2)$. Now we improve these results and present logspace uniform circuits that have size $O(M(n))$ and depth $O(\log n \log\log n)$. Newton approximations of high degree are used to gain as many bits as possible in the early stages, and thus reduce the overall number of stages required. An important property of the new algorithm is that the size bound of our circuit is asymptotically tight (within a constant factor) with the optimal size bound of multiplication, so further improvements in multiplication would be mirrored by improvements in integer division. Furthermore, by a classic result given in Aho,

Hopcroft, and Ullman [1], multiplication can be done with a constant number of reciprocals, so our circuit has optimal size, while matching the best known depth bounds for logspace uniform circuits. A result of Alt [2] immediately applies to our results to give as a corollary that any rational function can be evaluated in $O(M(n))$ size and $O(\log n \log\log n)$ depth.

We will first show how to compute reciprocals of polynomials in size $O(PM(n))$ and depth $O(\log n \log\log n)$, where $PM(n)$ is the size complexity of $O(\log n)$ depth polynomial multiplication. The polynomial problem provides a good introduction to high-order iterative methods. High-order iterative methods date back to Euler; a general discussion of high-order iteration formulas can be found in Traub [13]. The method of using high-order Newton approximations is of independent interest as a way of efficiently using processors in a parallel system.

2. Algorithm overview. In Cook's reduction of division to multiplication, he used second-order Newton approximations with each successive stage dealing with twice the number of bits as its predecessor. The sequential complexity of a single stage of second-order Newton iteration is $O(SM(n))$. Since $SM(n)$ must be at least linear, the geometric progression of approximation lengths makes the sum over all stages no more than $O(SM(n))$. However, the circuit model of multiplication has size $M(n)$ and depth $O(\log n)$, and both size and depth must be summed over all stages. The same effect is noticed with the geometrically decreasing sizes, and the overall size of Cook's division algorithm is $O(M(n))$. Unfortunately, since the depth is only logarithmic, the fact that n is geometrically decreasing is not enough to keep the total depth from increasing to $\Omega(\log^2 n)$ in the summation.

Our key observation was that since the size and depth of the first stages in Cook's algorithm are so small, considerably more work can be done than a simple second-order approximation. Our algorithm consists of two parts: part A uses high-order Newton approximations, and part B extends this result to n bits using $O(\log\log n)$ second-order approximations. We present a formula for calculating the kth order Newton iteration for the reciprocal problem which increases the accuracy of an approximation (in bits) by a factor of k. In the early stages, k can be made large, so much more work can be done on each stage than simply doubling the number of bits as done by Cook.

The value of k for a particular stage is selected by making the size of every stage meet the same bound. The result is that the number of approximation stages required drops from $\Omega(\log n)$ to $O(\log\log n)$ for both integer reciprocals and polynomial reciprocals. The required number of iterations is heavily influenced by the size complexity of taking large powers, and for integer powering we present a new, size efficient powering algorithm.

3. High-order iterations for polynomial inverse. Let $R = \{D, +, \cdot, 0, 1\}$ be an arbitrary ring; we define a polynomial $p(x)$ of degree $n-1$ in $R[X]$ to be $p(x) = \sum_{i=0}^{n-1} a_i x^i$. In this section we will often define a new polynomial of degree $k-1$ by using the coefficients of the k highest degree terms of a higher-degree polynomial. The degree $k-1$ polynomial derived in this way from $p(x)$ is denoted by $p_k(x) = \sum_{i=0}^{k-1} a_{n-k+i} x^i$. In problems dealing with polynomials we use the bounded fan-in circuit model, but allow each node to compute either addition, multiplication, or reciprocation in the ring R in unit size and unit depth. Note that since reciprocation is allowed, some computations may be undefined.

The polynomial reciprocal problem as defined in Aho, Hopcroft, and Ullman [1]

is to calculate a polynomial $q(x)$ from a $(n-1)$st degree polynomial $p(x) \in R[X]$ such that

$$q(x) = \text{RECIPROCAL}(p(x)) = \left\lfloor \frac{x^{2n-2}}{p(x)} \right\rfloor .^{1} \tag{1}$$

It is easy to see that $q(x)$ must have degree $n-1$.

As previously mentioned, high-order iterative methods take an estimate of length d, and produce a new estimate of length kd. In the case of polynomials, we use $\text{RECIPROCAL}(p_d(x))$ as the length d "estimate" — note that $\text{RECIPROCAL}(p(x))$ can be written as $\text{RECIPROCAL}(p_n(x))$. To produce the estimate of length kd, first calculate the intermediate polynomial

$$r(x) = s(x) \sum_{j=0}^{k-1} \left[x^{(k+1)d-2}\right]^{k-j-1} \left[x^{(k+1)d-2} - p_{kd}(x)s(x)\right]^{j}, \tag{2}$$

where $s(x) = \text{RECIPROCAL}(p_d(x))$. Now let

$$q(x) = \left\lfloor \frac{r(x)}{x^{(k-1)(kd-2)}} \right\rfloor . \tag{3}$$

LEMMA 3.1. *Given* $s(x) = \text{RECIPROCAL}(p_d(x))$, *the polynomial* $q(x)$ *computed from* (3) *is exactly* $\text{RECIPROCAL}(p_{kd}(x))$; *furthermore,* $q(x)$ *can be computed in* $O(k^3 d \log(kd))$ *size and* $O(\log(kd))$ *depth.*

Proof. First we prove the correctness of the iteration formula (3). The lemma can be stated in a different (but equivalent) way; that is, that (3) produces a polynomial $q(x)$ such that $q(x)p_{kd}(x) = x^{2kd-2} + t(x)$ where $t(x)$ is some polynomial of degree less than $kd-1$. The polynomial $s(x)$ satisfies $p_d(x)s(x) = x^{2d-2} + t_1(x)$, where $\text{degree}[t_1(x)] < d-1$.

Since $p_{kd}(x) = p_d(x)x^{(k-1)d} + p'(x)$ (where $\text{degree}[p'(x)] \le d(k-1)-1$), multiplying by $s(x)$ gives $p_{kd}(x)s(x) = x^{(k+1)d-2} + x^{(k-1)d}t_1(x) + s(x)p'(x)$. For simplicity of notation, let $f(x) = x^{(k+1)d-2}$ and $g(x) = -(x^{(k-1)d}t_1(x) + s(x)p'(x))$, so $p_{kd}(x)s(x) = f(x) - g(x)$. Using this notation the iteration formula gives

$$\begin{aligned} p_{kd}(x)r(x) &= [f(x) - g(x)] \sum_{j=0}^{k-1} [f(x)]^{k-j-1}[g(x)]^{j} \\ &= \sum_{j=0}^{k-1} [f(x)]^{k-j}[g(x)]^{j} - \sum_{j=1}^{k} [f(x)]^{k-j}[g(x)]^{j} \\ &= [f(x)]^{k} - [g(x)]^{k} \\ &= x^{(k+1)kd-2k} - \left[-(x^{(k-1)d}t_1(x) + s(x)p'(x))\right]^{k}. \end{aligned}$$

Now doing the division by $x^{(k-1)(kd-2)}$ (which is actually just a shift of coefficients) and discarding the remainder, we get

$$p_{kd}(x)q(x) = x^{2kd-2} - \left\lfloor \frac{\left[-(x^{(k-1)d}t_1(x) + s(x)p'(x))\right]^{k}}{x^{(k-1)(kd-2)}} \right\rfloor .$$

[1] The floor function for the division of polynomials is analogous to the floor function applied to integers. In other words, in (1), $q(x)$ is the unique polynomial such that $x^{2n-2} = q(x)p(x) + r(x)$, where $r(x)$ is the remainder and satisfies $\text{degree}[r(x)] < \text{degree}[p(x)]$.

To simplify notation, let

$$t(x) = -\left\lfloor \frac{\left[-(x^{(k-1)d}t_1(x) + s(x)p'(x))\right]^k}{x^{(k-1)(kd-2)}} \right\rfloor,$$

so $p_{kd}(x)q(x) = x^{2kd-2} + t(x)$.

Examining the degrees of the components of $t(x)$ we see that the numerator is just $g(x)$ which, on closer examination, satisfies degree$[g(x)] \leq kd - 2$. After the powering of $g(x)$ we see that degree$[g(x)]^k \leq k^2d - 2k$. The division gives $t(x)$ with degree$[t(x)] \leq kd - 2$, and the correctness of our formula is proved.

To determine the complexity of the iteration formula, first note that a polynomial of degree $kd - 1$ can be raised to the kth power in size $O(k^2d\log(kd))$ and depth $O(\log(kd))$ by using discrete Fourier transforms (see, for example, [9]). There are k different powers to take and add up, and this cost dominates the entire calculation. The total size is $O(k^3d\log(kd))$, and the total depth is $O(\log(kd))$. □

We repeatedly apply the iteration formula of (3) to get the complete polynomial reciprocal algorithm. The details are described in the proof of the following theorem.

THEOREM 3.2. *The reciprocal of an $(n-1)$st degree polynomial $p(x)$ as defined above can be computed in $O(PM(n))$ size and $O(\log n \log\log n)$ depth, where $PM(n)$ is the size complexity of $O(\log n)$ depth polynomial multiplication.*

Proof. Without loss of generality, we can assume that n is a power of two as explained in Aho, Hopcroft, and Ullman [1]; in particular, we let $n = 2^m$ for some integer m. Let $f(i) = \lceil m(1 - (2/3)^{i-1})\rceil$; now we can define a sequence of values by $d_i = 2^{f(i)}$. Note that $d_1 = 1$. Letting $p(x) = a_n x^{n-1} + p'(x)$, then a_n^{-1} is the reciprocal of the degree 0 polynomial that serves as the base of our algorithm.

The order of the iteration formula that we use at stage i is $k_i = 2^{f(i+1)-f(i)}$, and it is easy to show that $f(i+1) - f(i) \leq (m/3)\,(2/3)^{i-1} + 1$. Substituting actual values for d_i and k_i in the complexity bounds, stage i takes size $O(n\log n)$ and depth $O(\log n)$. The number of iterations is $O(\log\log n)$ (this is easy to see — it can be verified by solving $m(1 - (2/3)^{i-1}) = m - 1$), so the total size of all stages of this algorithm is $O(n\log n\log\log n)$, and the total depth is $O(\log n\log\log n)$.

However, the algorithm that was just described is not quite what we use. If we take the first $n/\log^2 n$ coefficients of $p(x)$ and find the reciprocal of the polynomial defined by these coefficients, then letting $n' = n/\log^2 n$ the previously mentioned algorithm takes size $O(n'\log n'\log\log n') = O(n)$ and depth $O(\log n\log\log n)$. This is part A of our polynomial reciprocal algorithm.

Part B is a series of second-order iterations (using the result of part A as an initial estimate), and the required number of stages is $O(\log\log n)$. Part B is easily seen to have size $O(PM(n))$ and depth $O(\log n\log\log n)$. The total complexity of our polynomial reciprocation algorithm is the sum of parts A and B, so the total size is $O(PM(n))$, and the total depth is $O(\log n\log\log n)$. □

4. Calculating integer powers. At the heart of our circuit for integer reciprocals is an improved modular powering algorithm based on previous results of Reif [9], and Shankar and Ramachandran [12]. Both previous algorithms use divide and conquer (by Discrete Fourier Transform) to work on powering problems with smaller numbers in parallel. For our division circuit, we need a circuit for mth order Newton iteration of an n bit number that has size $O(nm^{O(1)}(\log n)^{O(1)})$. Unfortunately, the powering algorithm in Reif [9], has size that is quadratic in n, and the circuit of

Shankar and Ramachandran [12], though it improves the size bounds, also has size that grows too fast in n for our intended application.

The divide-and-conquer approach of Shankar and Ramachandran [12], reduces the number of bits at each stage, but the power remains the same throughout. In our algorithm, we reduce *both* the number of bits and the power at each stage. Note that to raise an n bit number x to the mth power, where m is a perfect square, we can first raise x to the $\sqrt{m}$th power, and then raise this result to the $\sqrt{m}$th power. Unfortunately, m is often not a perfect square, so let the first power be $p_1 = \lfloor \sqrt{m} \rfloor$. Next, see if $p_1(p_1+1) \leq m$, and if it is, the second power we take will be $p_2 = p_1 + 1$; if $p_1(p_1+1) > m$, then the second power will be just $p_2 = p_1$. The number x is first raised to the p_1th power, and this result is then raised to the p_2th power; the final result is $x^{p_1 p_2}$. Now since $p_1 p_2$ will usually not be m, we need to calculate an error term $e = m - p_1 p_2$. If we take x^e and multiply by the preceding result, the result is the desired answer of x^m. A simple calculation shows that $e < \sqrt{m}$, so the original powering problem has been reduced to three smaller powerings, each of size $\approx \sqrt{m}$. Note that the calculation of $x^{p_1 p_2}$ can happen in parallel with the calculation of x^e, so the depth is only that of two smaller powerings (not three).

By reducing powers in this way and reducing the number of bits of each subproblem with discrete Fourier transforms, the size of the problem is reduced very quickly. For the depth bounds to work out as we needed, it was discovered that the number of bits should decrease faster than the powers. To achieve this, the power is reduced only half as often as the number of bits. We will consider a stage of reducing both power and bits followed by a second stage of reducing only the number of bits as a single level in our circuit. Notice that the stage of reducing only the number of bits is exactly the circuit of Shankar and Ramachandran.

The powering algorithm can be found in pseudocode in Fig. 1. The recursive call to $\text{MODPOWER}_{\text{SR}}$ actually does a stage of the Shankar and Ramachandran circuit before recursively calling MODPOWER.

As shown in Reif [9], (also see Shankar and Ramachandran [12]), there will be no error with this algorithm as long as $2m(l+1+\log k) \leq k-1$, which is satisfied whenever $m \geq 32$ and $n > m^2$. A simple check shows that at all levels of our algorithm, these inequalities hold.

THEOREM 4.1. *The circuit that calculates* $\text{MODPOWER}(\cdot, m, n)$ *with the constraint* $m \leq \sqrt{n}$ *has size* $O(nm^4 \log n \log\log n)$, *and depth* $O(\log n + \log m \log\log m)$; *furthermore, the circuit is logspace constructible.*

Proof. We will use the notation $S(n,m)$ and $T(n,m)$ to denote the size and depth, respectively, of taking the mth power of an n bit number modulo 2^n+1 using our MODPOWER algorithm. The MODPOWER algorithm deals only with integer values, and consequently, floors and ceilings are often taken. These are analyzed by repeatedly applying the following inequality — when bounding a product such as $\lceil m \rceil \lceil n \rceil$, note that

$$\lceil m \rceil \lceil n \rceil < (m+1)(n+1) < mn\left(1 + \frac{1}{m} + \frac{1}{n} + \frac{1}{mn}\right). \tag{4}$$

If there is a constant lower bound for m and n, then $\lceil m \rceil \lceil n \rceil$ can be bounded by $\lceil m \rceil \lceil n \rceil < cmn$ for some constant c (in many cases below, we actually bound the constant c).

We first derive a recurrence equation for the size of the MODPOWER circuit. In the pseudocode, lines marked with an asterisk (*) take no size or depth in the circuit

MODPOWER(x, m, n) /* Calculate $x^m \bmod 2^n + 1$ */
  if $m < 32$ **then**
(1) Calculate using Schönhage-Strassen multiplication algorithm.
  else
(*) $k \leftarrow \lceil \sqrt{nm} \rceil$
(*) $l \leftarrow \lfloor \frac{n}{k} \rfloor$
(*) $p_1 \leftarrow \lfloor \sqrt{m} \rfloor$
(*) **if** $p_1(p_1 + 1) \leq m$ **then**
(*) $p_2 \leftarrow p_1 + 1$
(*) **else**
(*) $p_2 \leftarrow p_1$
(*) **fi**
(*) $e \leftarrow m - p_1 p_2$
  In parallel do part1, part2
    part1:
(2) $t \leftarrow$ MPMACRO(x, e)
    part2:
(3) $y \leftarrow$ MPMACRO(x, p_1)
(4) $z \leftarrow$ MPMACRO(y, p_2)
  od
(5) MODPOWER $\leftarrow zt \bmod 2^n + 1$
  fi

$y' \leftarrow$ MPMACRO(x', m') /* Uses k, l, and n from above */
  $y' \leftarrow x'$
(1) Divide y' into k blocks of l bits each, such that $y' = \sum_{i=0}^{k-1} y_i 2^{il}$ and
  for all i, $0 \leq y_i < 2^l$
(2) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow (y_0, 2^1 y_1, 2^2 y_2, \cdots, 2^{k-1} y_{k-1}) \bmod 2^k + 1$
(3) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow \mathrm{DFT}_k(y_0, y_1, y_2, \cdots, y_{k-1}) \bmod 2^k + 1$
  In parallel for $i = 0, 1, \cdots, k-1$ **do**
(4) $y_i \leftarrow$ MODPOWER$_{\mathrm{SR}}(y_i, m', k)$ /* Uses ([12]) */
  od
(5) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow \mathrm{DFT}_k^{-1}(y_0, y_1, y_2, \cdots, y_{k-1}) \bmod 2^k + 1$
(6) $(y_0, y_1, y_2, \cdots, y_{k-1}) \leftarrow (y_0, 2^{-1} y_1, 2^{-2} y_2, \cdots, 2^{-(k-1)} y_{k-1}) \bmod 2^k + 1$
(7) $y' \leftarrow y_0 + y_1 2^l + y_2 2^{2l} + \cdots + y_{k-1} 2^{(k-1)l} \bmod 2^n + 1$

FIG. 1. *Pseudocode for* MODPOWER.

— they are calculated when the circuit is constructed. Ignoring the case of line (1) for now, we see that all lines other than those calling MPMACRO take total size $O(M(n))$ and total depth $O(\log n)$.

Deriving the size of MPMACRO can be done as follows. Assuming that $m > 32$ (so there is at least one level of recursion), let $k_1 = \lceil \sqrt{nm} \rceil$ be the k from MODPOWER, and let $k_2 = \lceil 2\sqrt{k_1 m'} \rceil$ be the k from the application of the Shankar and Ramachandran circuit. All steps except line (4) are easily done in size $O(k_1^2 \log k_1)$. Line (4) includes a stage of the Shankar and Ramachandran circuit as described in

the text preceding the theorem. For each i, the size of this reduction is bounded by $k_2 S(k_2, m') + ck_2^2 \log k_2$ for some constant c. As there are k_1 different values of i for line (4), the total size of MPMACRO is bounded by $k_1 k_2 S(k_2, m') + ck_1 k_2^2 \log k_2$.

Noting that $m' \leq \lceil m^{1/2} \rceil$ and that MPMACRO is called three times, we get the following recurrence equation for the size of MODPOWER:

$$S(n,m) = \begin{cases} c_1 M(n) & \text{if } m \leq 32, \\ 3k_1 k_2 S(k_2, \lceil m^{1/2} \rceil) + c_2 k_1 k_2^2 \log k_2 + c_3 M(n) & \text{otherwise.} \end{cases}$$

The claim is that for some constant c, $S(n,m) \leq cnm^4 \log n \log\log n$ satisfies this for all $m \leq \sqrt{n}$. For $m \leq 32$, this is obviously true.

For $m > 32$ but $\sqrt{m} \leq 32$, the recursive cost is given by the top line of the recurrence equation. Therefore, the total size is bounded by $3k_1 c_1 M(k_1) + c_2 k_1 \log k_1 + c_3 M(n)$. But $k_1 < n$ and $3c_1 k_1^2 < c_4 mn$ for some constant c_4, so this is bounded by $(c_1 + c_3)M(n) + c_2 n \log n$ — and it follows that the size claim holds.

For $\sqrt{m} > 32$, we need to look more closely at k_1 and k_2. Expanding k_2, we see that $k_2 = \left\lceil 2\left(\lceil \sqrt{nm} \rceil \lceil \sqrt{m} \rceil\right)^{1/2} \right\rceil$. Using the technique above for bounding products of ceilings, we bound $k_2 < \lceil 2.04 n^{1/4} m^{1/2} \rceil$. Using the same method, we see that $k_1 k_2 < 2.05 n^{3/4} m$. Using these facts, if $\sqrt{m} > 32$, then

$$S(n,m) \leq 6.15 n^{3/4} m S\left(\left\lceil 2.04 n^{1/4} m^{1/2} \right\rceil, \left\lceil m^{1/2} \right\rceil\right) + c_5 nm^{3/2} \log n + c_3 M(n).$$

Using our claim on the right-hand side and repeatedly using the bound from equation (4) for bounding products of ceilings, the size claim can be proved.

The depth of MPMACRO is even easier to compute than the size. The depth of all nonrecursion lines is $O(\log k_1)$, and there is a single recursion for a total depth bound of $T(k_2, m') + c \log k_1$.

Noting that MPMACRO gets called twice sequentially (lines (4) and (5)), the total depth of MODPOWER is bounded by $2T(k_2, m') + c \log n$. Using the bounding equations calculated above, the recurrence equations for the depth are

$$T(n,m) = \begin{cases} c_1 \log n & \text{if } m \leq 32, \\ 2T\left(\left\lceil 2.04 n^{1/4} m^{1/2} \right\rceil, \left\lceil m^{1/2} \right\rceil\right) + c_2 \log n & \text{otherwise.} \end{cases}$$

Our claim is that for some constant c, $T(n,m) \leq c(\log n + \log m \log\log m)$ satisfies the above equation. Again, if $m \leq 32$, there is nothing to prove.

If $m > 32$ but $\sqrt{m} \leq 32$, then there is just the one recursive call as in the size analysis. Since $\log k_2 < \log n$, the recurrence equation for the depth becomes $T(n,m) \leq (2 + c_2) \log n$ — therefore, setting $c = 2 + c_2$ is sufficient to prove the claim.

If $\sqrt{m} > 32$, it is important to note the following two inequalities:

$$\log \left\lceil 2.04 n^{1/4} m^{1/2} \right\rceil < \frac{1}{4} \log n + \frac{1}{2} \log m + 1.2,$$

$$\log \left\lceil m^{1/2} \right\rceil \log\log \left\lceil m^{1/2} \right\rceil < \frac{1}{2} \log m \log\log m + 0.06 \log\log m - 0.48 \log m - 0.05.$$

Using these values to put the claim in the right-hand side of the recurrence equations results in a proof that the claim holds for $c = 5c_2$.

As for the circuit being logspace constructible, it should be noted that all calculations made in the construction of the circuit (the lines marked with (*) in Fig. 1)

deal with numbers that are $O(\log n)$ bits long. In other words, these calculations only need to be done in space *linear* in the length of the numbers used — this is, of course, easily done. □

COROLLARY 4.2. MODPOWER *can compute* x^m, *where* x *is an* n *bit number and* $m \leq \sqrt{n}$, *in size* $O(nm^5 \log n \log\log n)$, *and depth* $O(\log n + \log m \log\log m)$. *This circuit is also logspace constructible.*

Proof. Simply use the modular powering algorithm of Theorem 4.1 to calculate $x^m \bmod 2^{nm}+1$. This ring is large enough to hold the exact answer, so the modular result will be the same as the exact result. □

5. High-order iteration for integer division. The following definition is useful when describing the amount of error present in an approximation.

DEFINITION. An approximation $\tilde{x}$ to a value x is said to be accurate to c bits if $|x - \tilde{x}| \leq 2^{-c}$.

Note that this definition is the intuitive definition of "accurate to c bits in the fractional part." The reciprocal problem is that given a value x, we need to find the value $y = 1/x$ to within a certain error bound. We will scale the input so that $\frac{1}{2} < x \leq 1$, which has no effect on the problem — the result will simply be scaled back at the end. The complexity is also not affected since the scaling can be done by powers of two (which can be done by bit shifting). If the scaled value of x is accurate to n bits, then we want y accurate to n bits.

Newton iteration is a general method of refining a guess to the exact answer of a problem of the form "find x such that $f(x) = 0$" for some given function f. The second-order Newton iteration formula for finding reciprocals has been known and used for quite some time (see, for example, [5]). What we use in this paper are Newton iterations of higher degree. In general, a kth order Newton iteration for the reciprocal problem is given by

$$y_{i+1} = y_i \sum_{j=0}^{k-1} (1 - xy_i)^j,$$

where the values y_i are the approximations to y.

In the following error analysis, let $\epsilon_{y,i}$ be the difference between y and the approximation y_i at step i, so $y_i = y - \epsilon_{y,i}$.

THEOREM 5.1. *If the error at step* i *is* $\epsilon_{y,i}$, *then after applying a* k*th order Newton iteration, the error at step* $i+1$ *satisfies the inequality* $|\epsilon_{y,i+1}| \leq |\epsilon_{y,i}|^k$.

Proof. Rewriting y_i as $y - \epsilon_{y,i}$, the Newton sum can be rewritten:

$$y_{i+1} = (y - \epsilon_{y,i}) \sum_{j=0}^{k-1} (1 - x(y - \epsilon_{y,i}))^j = (y - \epsilon_{y,i}) \sum_{j=0}^{k-1} (x\epsilon_{y,i})^j$$

since $xy = 1$. Further simplifications give

$$\begin{aligned} y_{i+1} &= y \sum_{j=0}^{k-1} (x\epsilon_{y,i})^j - \epsilon_{y,i} \sum_{j=0}^{k-1} (x\epsilon_{y,i})^j \\ &= y + \sum_{j=0}^{k-2} x^j \epsilon_{y,i}^{j+1} - \sum_{j=0}^{k-1} x^j \epsilon_{y,i}^{j+1} \\ &= y - x^{k-1} \epsilon_{y,i}^k. \end{aligned}$$

Since $x \leq 1$, this implies that $|\epsilon_{y,i+1}| \leq |\epsilon_{y,i}|^k$. □

In our algorithm we will use only iterations of even degree because of the nice ordering properties of even degree approximations. The following obvious corollary shows the relationship between y_{i+1} and y.

COROLLARY 5.2. *If k is even, then after applying a kth degree Newton iteration at step i, $y_{i+1} \leq y$.*

In the discussion above, we assumed that calculations were performed with all the bits of x (i.e., x has infinite precision). A natural question to ask is how many bits of x we really need to consider to achieve the desired error bound of $|\epsilon_{y,i+1}| \leq |\epsilon_{y,i}|^k$. We answer this question in the remainder of this section.

First, let us introduce some more notation. We will be taking only the most significant bits of x and throwing away the least significant bits. The truncated value is called $\tilde{x}$, and $\tilde{y} = 1/\tilde{x}$. It is trivial to see that $\tilde{x} \leq x$, so $\tilde{y} \geq y$. Let $\epsilon_{\tilde{x}} = x - \tilde{x}$, and $\epsilon_{\tilde{y}} = \tilde{y} - y$.

LEMMA 5.3. *If $|\epsilon_{y,i}| \leq c \leq \frac{1}{4}$ for some value c, and can insure that $\epsilon_{\tilde{y}} \leq c^k$, then performing the kth order iteration (k even) using $\tilde{y}$ will result in $|\epsilon_{y,i+1}| \leq c^k$.*

Proof. It is important to note that we are doing the exact Newton iteration for $\tilde{y}$. There are three cases to consider, one for each possible ordering of y, $\tilde{y}$, and y_i.

Case 1. $y \leq \tilde{y} < y_i$. As we noted in the preceding corollary, after performing the iteration $y_{i+1} \leq \tilde{y}$. Since $\tilde{y} - y \leq c^k$ and $\tilde{y} - y_{i+1} \leq |\epsilon_{y,i}|^k \leq c^k$, it follows that $|y - y_{i+1}| \leq c^k$.

Case 2. $y \leq y_i \leq \tilde{y}$. After the Newton iteration the order must be $y \leq y_{i+1} \leq \tilde{y}$, and since $\tilde{y} - y \leq c^k$, then $|y - y_{i+1}| \leq c^k$.

Case 3. $y_i < y \leq \tilde{y}$. After the Newton iteration either $y \leq y_{i+1} \leq \tilde{y}$ (and $|y - y_{i+1}| \leq c^k$ as in Case 2), or $y_{i+1} < y \leq \tilde{y}$. Considering the latter ordering, $\tilde{y} - y_i = \epsilon_{y,i} + \epsilon_{\tilde{y}}$, so $\tilde{y} - y_{i+1} \leq (\epsilon_{y,i} + \epsilon_{\tilde{y}})^k$ and $y - y_{i+1} \leq (\epsilon_{y,i} + \epsilon_{\tilde{y}})^k - \epsilon_{\tilde{y}}$. Furthermore,

$$\begin{aligned}(\epsilon_{y,i} + \epsilon_{\tilde{y}})^k - \epsilon_{\tilde{y}} &= \sum_{j=0}^{k} \binom{k}{j} \epsilon_{\tilde{y}}^{j} \epsilon_{y,i}^{k-j} - \epsilon_{\tilde{y}} \\ &= \epsilon_{y,i}^{k} + \epsilon_{\tilde{y}} \left(\sum_{j=1}^{k} \binom{k}{j} \epsilon_{\tilde{y}}^{j-1} \epsilon_{y,i}^{k-j} - 1 \right).\end{aligned}$$

Now look at the sum

$$\begin{aligned}\sum_{j=1}^{k} \binom{k}{j} \epsilon_{y,i}^{k-j} \epsilon_{\tilde{y}}^{j-1} &\leq \sum_{j=1}^{k} \binom{k}{j} c^{k-j} (c^k)^{j-1} \\ &= \sum_{j=1}^{k} \binom{k}{j} c^{j(k-1)} \\ &< c^{k-1} \sum_{j=1}^{k} \binom{k}{j} \\ &\leq 2^{-2(k-1)} (2^k - 1) \\ &= 2^{-k+2} - 2^{-2(k-1)} \\ &\leq 1 \quad \text{for all } k \geq 1.\end{aligned}$$

This implies that $y - y_{i+1} \leq \epsilon_{y,i}^k \leq c^k$, and $|y - y_{i+1}| \leq c^k$. □

The following theorem sums up the point of the entire section.

THEOREM 5.4. *If y_i is accurate to p bits with $p \geq 2$, then applying a kth order Newton iteration (where k is even) using the first $kp+2$ bits of x results in y_{i+1} accurate to kp bits.*

Proof. y_i is accurate to $p \geq 2$ bits means that $|\epsilon_{y,i}| \leq 2^{-p} \leq \frac{1}{4}$. Let $c = 2^{-p}$ and note that $\epsilon_{\tilde{x}} \leq 2^{-(kp+2)} = \frac{1}{4}c^k$. Now look at $\epsilon_{\tilde{y}}$.

$$\epsilon_{\tilde{y}} = \frac{1}{\tilde{x}} - \frac{1}{x} = \frac{x - \tilde{x}}{\tilde{x}x} = \frac{x - (x - \epsilon_{\tilde{x}})}{\tilde{x}x} = \frac{\epsilon_{\tilde{x}}}{\tilde{x}x} .$$

Since $x \geq \tilde{x} \geq \frac{1}{2}$, we know that $\epsilon_{\tilde{y}} \leq 4\epsilon_{\tilde{x}} \leq c^k$. Now Lemma 5.3 directly applies to give $|\epsilon_{y,i+1}| \leq c^k = 2^{-kp}$, so y_{i+1} is accurate to kp bits. □

6. The complexity of each step. In this section we derive size and depth bounds for refining a p bit approximation to pk bits. As seen in the previous section, a kth degree Newton iteration (assume k is even from here on) on a p bit approximation yields a new approximation of at least pk bits when the first $pk+2$ bits of x are used. Therefore we first determine the complexity of a kth order Newton iteration, using pk bits, then see what happens when two more bits are used.

To calculate the required approximations, we use the new method of powering introduced in § 4 to obtain the following results.

THEOREM 6.1. *The kth order Newton iteration of a p bit number (using pk bit calculations and giving a pk bit result) can be computed by a logspace uniform circuit family of size $O(pk^7 \log pk \log\log pk)$, and depth $O(\log p + \log k \log\log k)$.*

Proof. Looking at the Newton iteration formula of § 5, we first need to calculate $u = 1 - xy_i$. This can easily be done in $O(M(pk))$ size and $O(\log pk)$ depth. Next, we need to calculate u^i for $0 \leq i < k$, which is done by the circuit of § 4. The powers of u are then all added together with size $O(pk^2)$ and depth $O(\log pk)$, and the final multiplication by y_i is performed. Clearly, the cost of performing the k powerings dominates the entire circuit, so the total size is $O(pk^7 \log pk \log\log pk)$, and the depth is $O(\log pk + \log k \log\log k) = O(\log p + \log k \log\log k)$. □

It is important to note that the summation in the Newton iteration formula is a simple truncated power series and can be factored in exactly the same manner as the reciprocal power series in Melhorn and Preparata [7] and Shankar and Ramachandran [12]. After such a factoring, the largest power that needs to be taken is k^ϵ for some constant $\epsilon > 0$, and the resulting circuit has size $O(pk^{1+6\epsilon} \log pk \log\log pk)$ while the depth remains essentially unchanged. Setting $\epsilon = \frac{1}{6}$, we get the following corollary.

COROLLARY 6.2. *The kth order Newton iteration of a p bit number (using pk bit calculations) can be calculated by a logspace uniform circuit family of size $O(pk^2 \log pk \log\log pk)$ and depth $O(\log p + \log k \log\log k)$.*

The calculations that follow do not guarantee that k is an integer. In such a case, we perform an order $\lceil k \rceil$ Newton iteration, which will produce an approximation accurate to at least pk bits. Adding one or two to k, if needed to take the ceiling and make it even, obviously does not affect the asymptotic bounds. Similarly, doing calculations with $pk+2$ bits does not affect the asymptotic bounds. From these facts, Corollary 6.2 and Theorem 5.4, we get the following corollary.

COROLLARY 6.3. *An approximation accurate to pk bits can be obtained from a p bit approximation by a logspace uniform circuit of size $O(pk^2 \log pk \log\log pk)$ and depth $O(\log p + \log k \log\log k)$.*

7. The integer reciprocal algorithm. In this section, we get to the heart of the reciprocal algorithm. As mentioned in the overview of the algorithm, in part A of our algorithm we choose the highest degree Newton approximation possible, while staying within given size bounds. Let p_i denote the number of bits of accuracy at stage i, and define a sequence of accuracies by $p_i = n^{1-(1/2)^i}$; note that $p_0 = 1$ (only one bit needs to be known initially).

THEOREM 7.1. *Part* A *of the reciprocal algorithm calculates the reciprocal of* x *accurate to* $n/(\log n)^2$ *bits in* $O(n)$ *size and* $O(\log n \log\log n)$ *depth.*

Proof. From the formulas for p_i and p_{i+1}, we can easily solve to see what degree Newton iteration is needed at stage i — call this k_i:

$$k_i = \frac{p_{i+1}}{p_i} = n^{1-(1/2)^{i+1}-(1-(1/2)^i)} = n^{(1/2)^{i+1}}$$

Now we can derive the size complexity of step i to be bounded by

$$\begin{aligned} cp_i k_i^2 \log p_i k_i \log\log p_i k_i &\le cn^{1-(1/2)^i} n^{(1/2)^i} \log n \log\log n \\ &\le cn \log n \log\log n. \end{aligned}$$

If we let $r = \log\log n$, then we see that $p_r = \frac{n}{2}$, so we know half of the bits. A single second-order Newton iteration extends this result to the full answer. Therefore, the total size for all r stages is $O(n \log n (\log\log n)^2)$.

Again (as in the polynomial reciprocal problem), we simply do not use all n bits for part A. If we let $N = n/(\log n)^2$, then performing the above algorithm on an N bit number produces a result accurate to N bits in size $O(N \log N (\log\log N)^2) = O(n)$.

The depth calculation is slightly more subtle. Looking at stage i, the depth of this stage is bounded by

$$c \log p_i + c \log k_i \log\log k_i \le c\left(\tfrac{1}{2}\right)^i \log n \log\log n + c \log n.$$

Summing over all r stages, and noting that $\sum \left(\frac{1}{2}\right)^i$ is bounded by a constant (it is bounded by 2, to be exact), the total depth is $O(\log n \log\log n)$. For the depth, decreasing the number of bits to N has no substantial effect, so the total depth is the same. □

Now we look at part B of the reciprocal algorithm, namely, using second-order Newton iterations to extend the approximation of part A to n bits.

THEOREM 7.2. *Part* B *of the reciprocal algorithm produces the reciprocal accurate to* n *bits from the result of part* A *in size* $O(M(n))$ *and depth* $O(\log n \log\log n)$.

Proof. If n_1 bits are known initially, then after applying m second-order Newton iterations, the approximation is extended to $n_2 = n_1 2^m$ bits. Using the number of bits produced by part A (Theorem 7.1) as n_1, letting $n_2 = n$, and solving for m, we get $m = 2 \log\log n$.

The size of second-order Newton iteration on n_i bits is less than $cM(n_i)$ for some constant c. The number of bits in the last stage is n, and for simplicity of notation we number the stages from the end with $n_0 = n$ and $n_i = n_{i-1}/2 = n/2^i$. The size of stage i is then less than $cM(n/2^i)$, which is less than $(c/2^i)M(n)$ since $M(n)$ must be at least linear. The sum over all stages is now easily evaluated as $\sum cM(n_i) \le 2cM(n)$, so the total size of part B is $O(M(n))$. The depth of each stage is $O(\log n)$, so the total depth of part B is $O(\log n \log\log n)$. □

Now we are ready to put both parts together and state size and depth bounds for the entire reciprocal circuit.

THEOREM 7.3. *The reciprocal of an n bit number can be calculated to n-bit precision by a logspace uniform circuit in size* $O(M(n))$ *and depth* $O(\log n \log\log n)$.

Theorem 7.3 is immediately applicable to other problems whose complexity is dominated by that of division. A *rational function* f is any function that can be written in the form $f(x) = p(x)/q(x)$, where p and q are fixed degree polynomials with coefficients that can be represented in fixed-point binary with $O(n)$ bits. In a recent paper, Alt [2] shows how multiplication is simultaneous size and depth equivalent to the evaluation of polynomials; therefore, in particular, the evaluation of $p(x)$ and $q(x)$ above can be reduced to multiplication. These results can be combined with a single division to produce $f(x)$, which gives rise to the following corollary.

COROLLARY 7.4. *Any rational function can be evaluated in* $O(\log n \log\log n)$ *depth and* $O(M(n))$ *size.*

8. Conclusion and open problems. The important contribution of this paper is that the size bounds for multiplication are matched by a division circuit with depth less than $\Omega(\log^2 n)$; in fact, we match the best known depth bounds for logspace uniform reciprocal circuits while obtaining optimal size. Note that if the size of multiplication (call this $M(n)$) is reduced, then using the new multiplication circuit in part B of our algorithm reduces the size of our division circuit to $O(M(n))$ also.

There are still interesting questions regarding the use of high-order Newton iterations. We know that all rational functions can be evaluated in identical bounds (by Corollary 7.4). This gives strong evidence that other algebraic problems can be solved using this technique.

An open question remaining in integer division is reducing the depth of the logspace uniform circuits. This seems to be a very hard problem requiring a different approach entirely.

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] H. ALT, *Comparing the combinatorial complexities of arithmetic functions*, J. Assoc. Comput. Mach., 35 (1988), pp. 447–460.

[3] 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.

[4] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

[5] S. A. COOK, *On The Minimum Computation Time of Functions*, Ph.D. thesis, Harvard University, Cambridge, MA, 1966.

[6] J. HASTAD AND T. LEIGHTON, *Division in* $O(\log n)$ *depth using* $O(n^{1+\epsilon})$ *processors*, unpublished note, 1986.

[7] K. MEHLHORN AND F. P. PREPARATA, *Area-time optimal division for* $T = \Omega((\log n)^{1+\epsilon})$, in Symposium on Theoretical Aspects of Computer Science, Lecture Notes in Computer Science 210, Springer-Verlag, New York, 1986, pp. 341–352.

[8] J. H. REIF, *Logarithmic depth circuits for algebraic functions*, in Proc. 24th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, D.C., 1983, pp. 138–145.

[9] ———, *Logarithmic depth circuits for algebraic functions*, SIAM J. Comput., 15 (1986), pp. 231–241.

[10] J. H. REIF AND S. R. TATE, *Efficient parallel integer division by high order Newton iteration*, preliminary draft, 1988.

[11] A. SCHÖNHAGE AND V. STRASSEN, *Schnelle Multiplikation grosser Zahlen*, Computing, 7 (1971), pp. 281–292.

[12] N. SHANKAR AND V. RAMACHANDRAN, *Efficient parallel circuits and algorithms for division*, Inform. Process. Lett., 29 (1988), pp. 307–313.

[13] J. F. TRAUB, *Iterative Methods for The Solution of Equations*, Chelsea, New York, 1964.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 925–967, October 1990

012

SORTING POINTS ALONG AN ALGEBRAIC CURVE*

JOHN K. JOHNSTONE† AND CHANDERJIT L. BAJAJ‡

Abstract. An operation that is frequently needed during the creation and manipulation of geometric models is the sorting of points along an algebraic curve. Given a segment $\overset{\frown}{AB}$ of an algebraic curve, a set of points on the curve is sorted from A to B along $\overset{\frown}{AB}$ by putting them into the order that they would be encountered in traveling continuously from A to B along $\overset{\frown}{AB}$. A new method for sorting points along a plane or space algebraic curve is presented. Key steps in this method are the decomposition of a plane algebraic curve into convex segments and point location in this decomposition. This new method can sort points on an arbitrary algebraic curve (including points spread over several connected components) and it is particularly efficient because of its preprocessing, both of which make it superior to conventional methods. The complexity of the new method is analyzed, and execution times of various sorting methods on a number of algebraic curves are presented. The theory developed for sorting can also be used to locate points on an arbitrary segment of an algebraic curve and to decide whether two points lie on the same connected component.

Key words. sorting, decomposition, point location, convexity, algebraic curves, geometric modeling, solid modeling

AMS(MOS) subject classifications. 68U05, 68Q25, 68P10, 14H99

1. Introduction.

The sorting of numbers into increasing order or words into alphabetical order is one of the basic problems of computer science. The purpose of this paper is to establish that the sorting of points along a curve is a basic problem in geometric modeling and computational geometry, and to present a universal and efficient method for this sorting. This method relies upon the solution of two problems that are very useful in their own right: convex decomposition of a curve and point location on a segment.

To sort a set of points from A to B along the curve segment $\overset{\frown}{AB}$ means to put the points into the order that they would be encountered in traveling continuously from A to B along $\overset{\frown}{AB}$ (Fig. 1). Points that do not lie on $\overset{\frown}{AB}$ are never encountered and are thus ignored. A tangent vector at A is provided to indicate the direction in which the sort is to proceed from A. This vector is especially important when the curve is closed, since there are then two segments between A and B from which to choose. All of the points, including A and B, are assumed to be nonsingular, since otherwise their order might be ambiguous.

Our treatment shall be of irreducible algebraic plane curves (a curve that lies in a plane and is described by an irreducible polynomial[1] $f(x, y) = 0$); in the rest of this paper, all curves are assumed to be of this type and nonlinear. An extension of the

* Received by the editors July 1, 1988; accepted for publication (in revised form) March 4, 1990.
† Department of Computer Science, The Johns Hopkins University, Baltimore, Maryland 21218.
The work of this author was supported in part by National Science Foundation grant IRI-8910366. While the author was a graduate student in the Department of Computer Science, Cornell University, he was supported by a Natural Sciences and Engineering Research Council of Canada 1967 Graduate Fellowship and an Imperial Esso Graduate Fellowship.
‡ Department of Computer Science, Purdue University, West Lafayette, Indiana 47907.
The work of this author was supported in part by National Science Foundation grant MIP-88-16286, Army Research Office contract DAAG 29-85-C0018 under Cornell MSI, and Office of Naval Research grant N00014-88-0402.

[1] The coefficient domain of the polynomial can be the integers, rationals, algebraic real numbers, or any other set of numbers that has a finite representation.

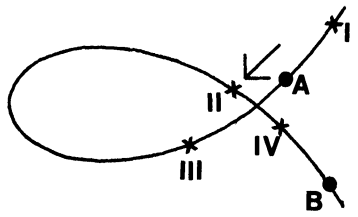


FIG. 1. *The sorted order from A to B is* III, II, IV.

methods to algebraic space curves is possible using a suitable projection of the space curve to a plane curve, as described in [18].

The next section establishes that sorting is a fundamental operation of geometric modeling. After discussing previous sorting methods in § 3, we introduce our new sorting method in § 4. Convex decomposition of a curve and point location on a convex segment are discussed in §§ 5 and 6. Complexity issues and execution times of the various sorting methods are presented in §§ 7 and 8, and § 9 provides some conclusions. The paper ends with an Appendix.

2. The importance of sorting. The sorting of points along a curve has many applications in geometric modeling. The following problem is the most natural application.

Restriction.
INSTANCE: A set S of points on a curve C and a segment $E_1\widehat{E_2}$ of C.
QUESTION: Which points of S lie on $E_1\widehat{E_2}$?
SOLUTION:

One solution is to sort $S \cup \{\text{endpoints } E_1, E_2\}$ along the curve and discard the points that do not lie between E_1 and E_2. However, a more efficient solution is simply to sort S along $E_1\widehat{E_2}$, starting at E_1. The output of this sort is the set of points on $E_1\widehat{E_2}$ (in sorted order). During this sort, the points of S that do not lie on $E_1\widehat{E_2}$ are either not encountered or they are encountered but eventually discarded. (This will be easier to understand after a description of the sorting algorithm.)

Since an edge of a solid model is often defined by a curve and a pair of endpoints, restriction is a very basic problem in geometric modeling. For example, the following edge intersection and bounding box problems are two important problems that can be solved with restriction.

Edge intersection.
INSTANCE: Edges E and F on curves C and D, respectively.
QUESTION: What is $E \cap F$?

SOLUTION:

Compute $C \cap D$ by well-known methods and restrict to the edges. That is, restrict $C \cap D$ to E and then restrict this $C \cap D \cap E$ to F.

Bounding box.

INSTANCE: Edge E on curve C with endpoints E_1 and E_2.

QUESTION: Find the smallest rectangle with sides parallel to the coordinate axes that contains E.

SOLUTION:

Compute the local extrema of the curve and restrict to the edge, yielding S. Find the minimum x-value ($x_{\min}$) in $S \cup \{E_1, E_2\}$, and so on. The desired box is defined by the lines $x = x_{\min}$, $x = x_{\max}$, $y = y_{\min}$, and $y = y_{\max}$.

The bounding box (see [22, p. 372]) is useful for interference detection: the expensive intersection of edges can be reserved for those situations when the edges are close enough that their bounding boxes interfere. Bounding regions are also useful for problems such as the restriction problem, because they allow points that clearly do not satisfy a condition to be discarded quickly.

Another fundamental use of sorting[2] is to introduce an even-odd parity to a set of points, as illustrated by the following problem.

Solid model intersection.

INSTANCE: Two solid models M and N.

QUESTION: What is the intersection of M and N?

SOLUTION:

An important step of this computation is to find the segments of an edge of one model that lie in the intersection. This is done by finding and sorting the points of intersection of this edge with a face of the other model. The segments of the edge between the ith and $(i+1)$st intersections, for i odd, are contained in the intersection of the models.

Another application of even-odd parity is to decide whether a point lies within a piecewise-algebraic plane patch (or a piecewise-algebraic convex surface patch). This problem, which is fundamental to the display of a geometric model, is fully discussed in [18]. Having established the importance of sorting, in the next section we proceed to a discussion of methods for sorting.

3. Previous work on sorting. There is no serious study of sorting in the literature. This can be explained by the fact that nontrivial sorting problems arise only with curves of degree three or more, and until recently, almost all of the curves in solid models were linear or quadratic. However, as the science of geometric modeling matures and grows more ambitious, curves of degree three and higher are becoming common. For example, the introduction of blending surfaces [17] into a model creates curves and surfaces of high degree.

[2] In this paper, "sorting" will always refer to the sorting of points along a curve, not the conventional sorting of numbers or words.

The lack of a study of sorting can also be explained by the presence of an obvious method for sorting points, which tends to obviate a search for any other method. This obvious method uses a rational parameterization of the curve (i.e., a parameterization $(x(t), y(t))$ such that both $x(t)$ and $y(t)$ can be expressed as the quotient of two polynomials in t), sorting a set of points S along $\overset{\frown}{AB}$ as follows.

The Parameterization Method of Sorting

[Preprocessing]

1. Parameterize the curve.

[Solve]

2. Find the parameter values of A and B, say t_1 and t_2.

3. Find the parameter value of each point in S.

[Sort numbers]

4. Sort the parameter values of S from t_1 to t_2, discarding those outside this interval.

We insist upon a rational parameterization because a nonrational parameterization is difficult to represent and difficult to solve. With a nonrational parameterization (such as $x(t) = \sqrt{t}$ or $x(t) = \sin(t)$), two different points may have the same parameter value, which complicates sorting. Finally, there is no algorithm for the automatic parameterization of a curve that does not have a rational parameterization, whereas there is such an algorithm for rational curves [2].

There are many reasons to be dissatisfied with the parameterization method. It is not a universal method, since not all algebraic curves have a rational parameterization. Indeed, a plane algebraic curve has a rational parameterization if and only if its genus is zero, if and only if it has the maximum number of singularities allowable for a curve of its degree [28]. Second, even for those curves that do have rational parameterizations, the parameterization method will be slow if the degree of the parameterization is high, since the computation of the parameter values of the points will be expensive. Other weaknesses of the parameterization method will become clear as we compare it with the new method, such as its difficulty with sorting points that are spread over several connected components of a curve (§ 6.3).

There is also a brute-force sorting method, which uses techniques for tracing along a curve [8]. The order of the points is the order in which they are encountered during a trace of the segment. This method is not satisfactory, because its implementation, although robust, is inherently very slow. Moreover, its complexity depends upon the length of the segment that is being sorted rather than upon the number of points in the sort, which is undesirable.

The weaknesses of the parameterization and tracing methods of sorting suggest that another method is necessary: one that will perform more efficiently on a wider selection of algebraic curves. The next section presents such a method. This method works with the implicit representation $f(x, y) = 0$ of a curve (as opposed to the parametric representation), thus allowing the use of tools from algebraic geometry.

4. The convex segment method of sorting. The observation that motivates the new method is that a convex segment can be sorted easily. Since every curve is a collection of convex segments, this suggests a divide-and-conquer strategy. A segment of a plane algebraic curve is *convex* if no line has more than two distinct intersections with it. Alternatively, a planar segment is convex if it lies entirely on one side of the

closed halfplane determined by the tangent line at any point of the segment [14]. The following theorem shows that sorting a convex segment is simple.

THEOREM 4.1. *Let $p_1, \cdots, p_n$ be points on a convex segment $\overset{\frown}{AB}$, and let H be the convex hull of A, B, $p_1, \cdots, p_n$ (Fig. 2). The order (from A to B) of $p_1, \cdots, p_n$ is simply the order (from A to B) of the vertices on the boundary of H.*

Proof. See p. 19 of [18] for the proof. □

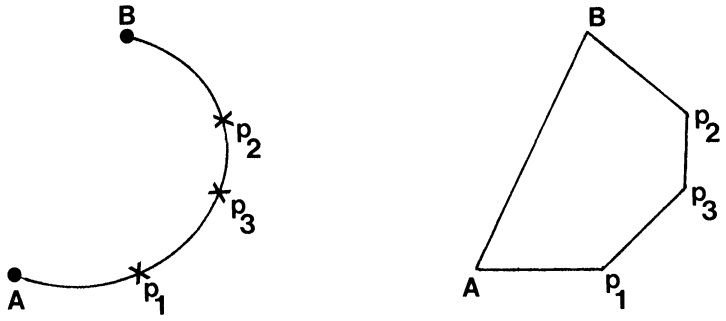


FIG. 2. *The sorting of a convex segment.*

Suppose that a curve can be decomposed into convex segments. Also suppose that the convex segments in this decomposition can be ordered so that it is simple to determine the predecessor and successor (if any) of each convex segment. Finally, suppose that, given a query point, we can identify the convex segment that contains the query point (point location in a convex decomposition). These key problems will be discussed in §§ 5 and 6. The following algorithm shows how to sort a set of points S along the segment $\overset{\frown}{AB}$.

The Convex Segment Method of Sorting

[Preprocessing]

1. Decompose the curve into convex segments (say S_1, S_2, $\cdots$, S_k).

[Locate first convex segment]

2. Find the convex segment that contains A (say $S_i = \overset{\frown}{W_1W_2}$).

3. Decide whether $\overset{\frown}{AB}$ leaves A along $\overset{\frown}{AW_1}$ or $\overset{\frown}{AW_2}$ (say $\overset{\frown}{AW_1}$).[3]

4. PresentConvSeg := $\overset{\frown}{AW_1}$; SortedSet := $\emptyset$; FoundB := false.

[Sort one convex segment at a time]

5. Repeat until FoundB

(a) Find the points of S that lie on PresentConvSeg.
If B is one of these points, then FoundB := true.

(b) Sort these points along PresentConvSeg, using Theorem 4.1.

---

[3] If V is the vector at A that is given as part of the input, then $\overset{\frown}{AB}$ leaves A along $\overset{\frown}{AW_1}$ if and only if V points to the halfplane defined by $\overset{\leftrightarrow}{AW_1}$ that contains $\overset{\frown}{AW_1}$.

(c) If not FoundB,
then SortedSet := Append(SortedSet,{sorted points on PresentConvSeg})
else SortedSet := Append(SortedSet,{sorted points on PresentConvSeg before B})

(d) PresentConvSeg := appropriate neighboring segment of PresentConvSeg

[Output]

6. Return SortedSet.

The expense of this method is concentrated in the preprocessing phase, which is done once off-line. The run-time operations (locating a point on a convex segment and sorting a set of points along a convex segment) are usually simple. Therefore, the efficiency of this method is very competitive. The coverage of the convex segment method is the entire set of algebraic curves, since it works directly from the implicit representation of the curve.

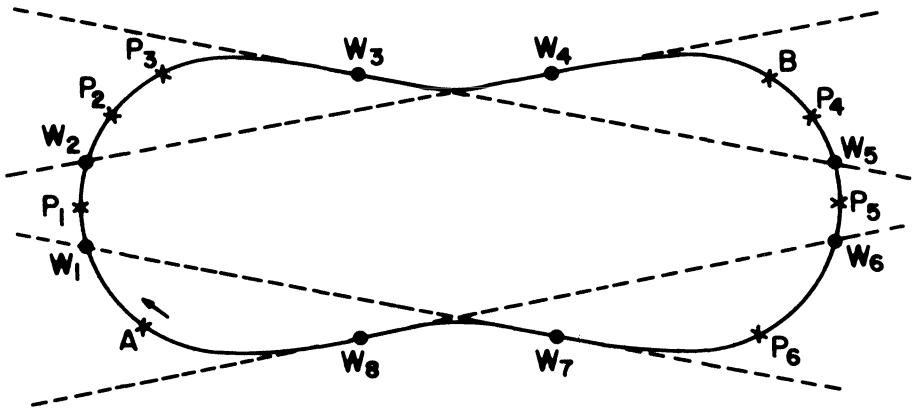


FIG. 3. *Sorting a curve by convex segments.*

Example 4.1. Consider the sorting of points $P_1, \cdots, P_6$ along the segment $\overset{\frown}{AB}$ of Fig. 3. The curve is decomposed into convex segments by the dotted lines (§ 5). A lies on $\overset{\frown}{W_1W_8}$ and the vector at A identifies that $\overset{\frown}{AW_1}$ is the first convex segment. There are no points on $\overset{\frown}{AW_1}$, so we move on. The next convex segment is $\overset{\frown}{W_1W_2}$. Only P_1 lies on $\overset{\frown}{W_1W_2}$ and it becomes the first element of the sorted list. We jump to the next convex segment $\overset{\frown}{W_2W_3}$ and sort the two points P_2 and P_3 by creating the convex hull of W_2, W_3, P_2, and P_3. P_2 and P_3 are added to the global sort. We move on to the next convex segment $\overset{\frown}{W_3W_4}$, and then $\overset{\frown}{W_4W_5}$. The presence of B indicates that this is the last convex segment. Upon sorting B and P_4, P_4 is discarded because it comes after B. The final sorted list is P_1, P_2, P_3.

It remains to discuss how a curve can be decomposed into convex segments and how a point can be located in this convex decomposition. These two problems, which are at the heart of the convex segment method of sorting, are solved in the following two sections.

5. Convex decomposition of a curve. The decomposition of an object into simple objects is an important theme in computational geometry. Decomposition proves to be particularly useful in divide-and-conquer algorithms, since simple objects are easily conquered. There has been a good deal of work on the decomposition of (simple, multiply connected, or rectilinear) polygons into simple components (e.g., triangles [12], [15], [16], [26], quadrilaterals [25], trapezoids [5], convex polygons [11], [27], and star-shaped polygons [6]), sometimes with added criteria (e.g., minimum decomposition [11], [19], minimum covering [23], no Steiner points [19]). However, all of this work has been in the polygonal (or at best polyhedral) domain. The decomposition of a plane algebraic curve of arbitrary degree into convex segments is an extension of decomposition to the curved world.

A version of Bezout's theorem states that two irreducible plane algebraic curves of degree m and n have exactly mn intersections (properly counted), unless the curves are identical [28]. Therefore, all plane algebraic curves of degree one (lines) and two (conics) are already convex. For the convex decomposition of curves of degree three and higher, the singularities and points of inflection are instrumental. A *singularity* of the curve $f(x, y) = 0$ is a point P of the curve such that $f_x(P) = f_y(P) = 0$ (where f_x is the derivative of f with respect to x). It is a point where the curve crosses itself or changes direction sharply. A nonsingular point is also called a *simple* point. A *point of inflection* is a simple point P of the curve whose tangent has three or more intersections with the curve at P. (It is also a point of zero curvature.) We restrict our attention to points of inflection P such that P's tangent has an odd number of intersections with the curve at P, which we call *flexes* for short. Fundamental in algebraic and differential geometry, singularities and flexes form a skeleton of the curve and can be used in many useful ways. (For example, singularities can be used to parameterize a plane algebraic curve [2].) Their use in convex decomposition underlines their importance to computational geometry of higher degrees.

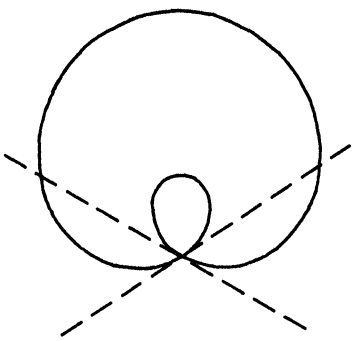


FIG. 4. *Convex segmentation of limacon of Pascal.*

The tangents at the singularities and flexes of a curve form an arrangement of lines that subdivide the plane of the curve into several regions (Figs. 3 and 4). The regions, which are closed and two-dimensional, are called *cells*, and the entire collection of cells is called a *cell partition* of the plane. For example, the cell partition defined by the curve of Fig. 4 consists of four unbounded cells. The line segments that bound a cell are called (cell) *walls* and are part of the cell.

The tangents also split the curve into several segments. The following theorem establishes that each of these segments is convex.

THEOREM 5.1. *The tangents of the singularities and flexes of a plane algebraic curve slice the curve into convex segments. That is, if $\overset{\frown}{PQ}$ is a nonconvex segment, then some tangent of a singularity or flex will intersect $\overset{\frown}{PQ}$.*[4]

Proof. Let $\overset{\frown}{PQ}$ be a nonconvex segment of an algebraic curve. Assume without loss of generality that $\overset{\frown}{PQ}$ does not contain a singularity or a flex. It can be shown that there exists a line L that crosses $\overset{\frown}{PQ}$ at three (or more) distinct points [18, p. 117].[5] Let x_1, x_2, and x_3 be three of these points, such that $x_2 \in \overset{\frown}{x_1x_3}$ and $\overset{\frown}{x_1x_3} \cap L = \{x_1, x_2, x_3\}$. $\overset{\frown}{x_1x_3}$ does not change its direction of curvature, since there is no singularity or flex on $\overset{\frown}{PQ}$. $\overset{\frown}{x_1x_3}$ is not a line segment; otherwise Bezout's theorem would imply that the algebraic curve that contains $\overset{\frown}{x_1x_3}$ is a line, which it cannot be since it contains a nonconvex segment. Therefore, it can be assumed without loss of generality that $\overset{\frown}{x_1x_3}$ looks like Fig. 5(a). Let R be the region bounded by $\overset{\frown}{x_1x_3}$ and $\overline{x_1x_3}$. We will show that R contains a singularity or a flex. This will complete the proof, since the tangent of a point inside R must intersect $\overset{\frown}{x_1x_3} \subset \overset{\frown}{PQ}$ at least once. (The tangent must cross the boundary of R twice, and at most one of these intersections can be with $\overline{x_1x_3}$.) The curve lies inside of R as it leaves $\overset{\frown}{x_1x_3}$ from x_1 and outside of R as it leaves $\overset{\frown}{x_1x_3}$ from x_3. Therefore, the curve must cross the boundary of R after it leaves $\overset{\frown}{x_1x_3}$ from x_1, either because it must join with x_3 (if the curve is bounded) or because an infinite segment of an algebraic curve cannot remain within a bounded region (if the curve is unbounded) [18, p. 116]. The curve cannot intersect the $\overset{\frown}{x_1x_3}$ boundary of R, since $\overset{\frown}{x_1x_3} \subset \overset{\frown}{PQ}$ is nonsingular by assumption. Therefore, the curve must cross $\overline{x_1x_3}$ after it leaves $\overset{\frown}{x_1x_3}$ from x_1.

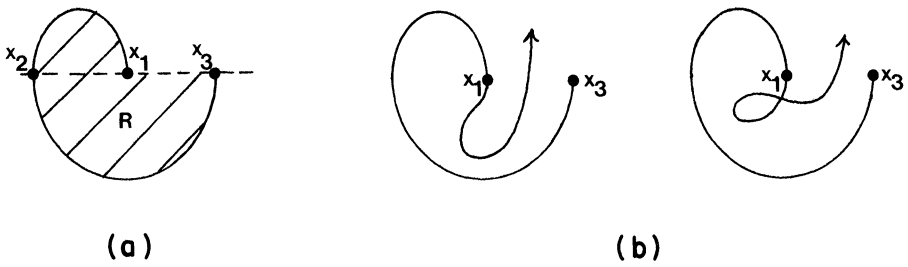


FIG. 5. (a) $\overset{\frown}{x_1x_3}$ *and* R. (b) *Traveling from* x_1 *to* $\overline{x_1x_3}$.

As the curve leaves $\overset{\frown}{x_1x_3}$ from x_1, it lies on the opposite side of x_1's tangent from $\overline{x_1x_3}$. Therefore, after the curve leaves $\overset{\frown}{x_1x_3}$ from x_1 and before it leaves R, the curve must cross x_1's tangent inside of R, in order to reach $\overline{x_1x_3}$. In order to cross over x_1's

[4] The simple points at which a singularity/flex tangent touches, but does not cross, the curve are redundant and should not be treated as convex segment endpoints in the decomposition.

[5] Already, by the definition of convexity, there must exist a line that intersects $\overset{\frown}{PQ}$ three (or more) times.

tangent, the curve must cross itself or change its curvature inside of R (Fig. 5(b)), otherwise it will spiral around inside R forever. Therefore, R contains a singularity or a flex. □

We include here a word about robustness. Consider the accuracy required in the computation of the singularities, flexes, and their tangents in order to guarantee a true division into convex segments. Suppose that, in the proof of Theorem 5.1, the tangent of a singularity/flex inside the region R is used to split a nonconvex segment. Any line through a point in the interior of R would work equally well in splitting the nonconvex segment. Thus, in this case the method is robust under slight errors in tangents, singularities, and flexes. The other case is if a nonconvex segment S is split into convex segments by a singularity or flex lying on S. The computed convex segment will differ from the actual convex segment by the same amount as the computed flex (say) differs from the actual flex. The only points that might be treated improperly are those that lie on the segment between the computed and actual flex. In other words, points that are within (some function of) machine precision of each other cannot be distinguished by the method and must be considered equivalent. This equivalence of points within machine precision is inherent to any sorting algorithm.

Theorem 5.1 does not solve the convex decomposition problem, because it yields a confused collection of endpoints of convex segments, not a collection of convex segments. The more challenging step of pairing up the endpoints remains, where two endpoints are *partners* if they define a convex segment of the decomposition. This pairing problem will be attacked in §§ 5.3 and 5.4, but first the collection of convex segments must be refined.

5.1. Refinement of convex segments I: Singularities. Many of the endpoints of the convex segments created by Theorem 5.1 are singularities. However, singular endpoints cause problems in pairing.[6] Consider a convex segment whose two endpoints are the same point, which might occur around a singularity (Fig. 4). This situation is to be avoided, since pairing will turn out to be easier if the two endpoints of a convex segment are different. It is also possible for a singularity to have more than two partners and, in particular, two partners in the same cell (e.g., singularity A in Fig. 7). This situation is also to be avoided, since it is easier to find the partner of an endpoint in a cell if this partner is unique. A third problem with singular endpoints is that the ordering of points about a singularity can be ambiguous. Does P_2 or P_3 follow A in Fig. 6? What is the order of the points on the loop of Fig. 6: S, P_1, P_2, P_3, S or S, P_3, P_2, P_1, S?

As a result of these problems, all convex segments with singular endpoints will be replaced by shorter segments with nonsingular endpoints. This is only a renaming procedure: although a segment $\overset{\frown}{AB}$ may be replaced by a segment $\overset{\frown}{A'B'} \subset \overset{\frown}{AB}$, $\overset{\frown}{A'B'}$ still represents the convex segment $\overset{\frown}{AB}$. This is best seen in the next phase, point location, where singular endpoints pose the same problems as they do in pairing endpoints. Although we work with $\overset{\frown}{A'B'} \subset \overset{\frown}{AB}$, if a point x lies on $\overset{\frown}{AB} \setminus \overset{\frown}{A'B'}$ (a segment of the curve that one might wrongly worry has been cut out of the curve), it is still located on the convex segment named $\overset{\frown}{A'B'}$.

[6] Unless otherwise noted, in the rest of the paper the term *endpoint* will be synonymous with "an endpoint of one of the convex segments created by the cell partition of a curve." Once we have discussed how to refine convex segments, we shall assume that all convex segments are refined and subsequently the term *endpoint* will refer to "an endpoint of one of the refined convex segments," while *original endpoint* will refer to an endpoint of a convex segment before refinement.

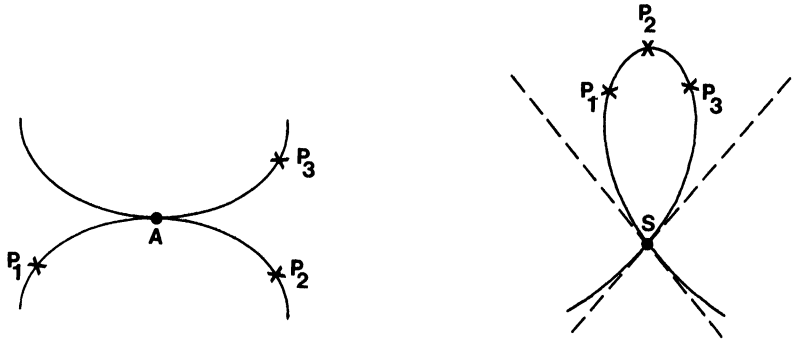


FIG. 6. *Ambiguity about a singularity.*

The endpoint A' that replaces A is called a *refined* endpoint, while A is called an *original* endpoint. A link is maintained between A' and A. Before refinement, a convex segment is called an *original* convex segment. After refinement (both of any singular endpoints and of any infinite parts as discussed in the next section), a convex segment is called a *refined* convex segment. Note that many segments will not require refinement, so their refined segments will be identical to their original segments.

The process of replacing convex segments with singular endpoints by convex segments with nonsingular endpoints is achieved by replacing singularities by nonsingular points. For each branch of the curve that passes through a singularity, a pair of nonsingular points are found: one on either side of (and usually close to) the singularity. As we shall see, the exact position of the nonsingular point is not important and is quite flexible.

Example 5.1. The original convex segments $\widehat{PA}$, $\widehat{AQ}$, $\widehat{RA}$, and $\widehat{AS}$ of Fig. 7 are replaced by the refined convex segments $\widehat{PV_1}$, $\widehat{V_2Q}$, $\widehat{RW_1}$, and $\widehat{W_2S}$. This is done by refining the singularity A into four points: V_1 and V_2 from one branch, and W_1 and W_2 from the other. A link is maintained between V_1 and V_2 (as well as between W_1 and W_2), so that it is clear that $\widehat{PV_1}$ is followed by $\widehat{V_2Q}$. Note that this refinement makes it clear that Q (not S) must follow P.

Consider the problem of finding two points on each branch of a singularity, one on either side of the singularity. We would like to do this by tracing [8] a small distance along the branch in both directions from the singularity. However, there is no reliable way of tracing along a branch as it passes through a singularity, because the other branches create too much confusion. Therefore, each branch of the singularity must be isolated so that it can be traced robustly. This isolation is accomplished by blowing up the curve at the singularity by a series of quadratic transformations [8], [28], [1], as follows. (Section 5.5, on the computation of singularities and flexes, is relevant to this discussion.)

The first step in blowing up a singularity is to translate it to the origin.[7] Let the

[7] Since the quadratic transformation does not map the line $x = 0$ properly, the curve should also be rotated if necessary so that it is not tangent to $x = 0$ at the origin (see [18]).

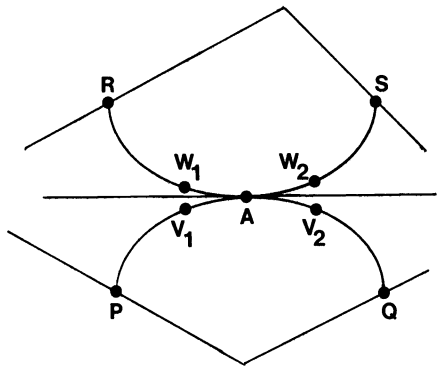


FIG. 7. *The refinement of a singularity.*

new equation of the curve be $f(x, y) = 0$. A quadratic transformation is now applied to the curve. The (*affine*) *quadratic transformation* $x = x_1$, $y = x_1 y_1$ [1] has three important properties:

- It maps the origin to the entire y_1-axis and the rest of the y-axis to infinity: $y_1 = y/x$ so $(0, b)$ maps to $(0, b/0)$, which is a point at infinity unless $b = 0$.
- It is one-to-one for all points (x, y) with $x \neq 0$.
- $y = mx$, a line through the origin, is mapped to the horizontal line $y_1 = m$: $y = mx \rightarrow x_1 y_1 = m x_1 \rightarrow y_1 = m$.

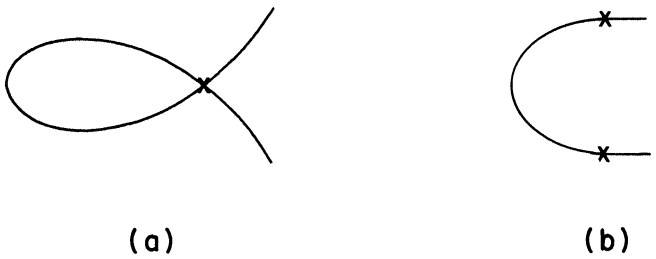


FIG. 8. (a) *Node.* (b) *Its quadratic transformation.*

Thus, a quadratic transformation maps distinct tangent directions of the various branches of f at the origin to different points on the *exceptional line* $x_1 = 0$. The intersections of the transformed branches with the exceptional line correspond to the transformed points of the origin (Fig. 8). If a point of $f(x_1, x_1 y_1)$ on the exceptional line is singular, then the procedure is applied recursively (Fig. 9). The following lemma establishes that the various branches of the curve in the neighborhood of the

singularity eventually get transformed to separate branches.

LEMMA 5.2 [1], [2], [28]. *A singularity can be reduced to a number of simple points by a finite number of applications of the quadratic transformation. An ordinary singularity can be reduced to simple points by a single quadratic transformation, where a singularity of multiplicity r is ordinary if its r tangents are all distinct.*

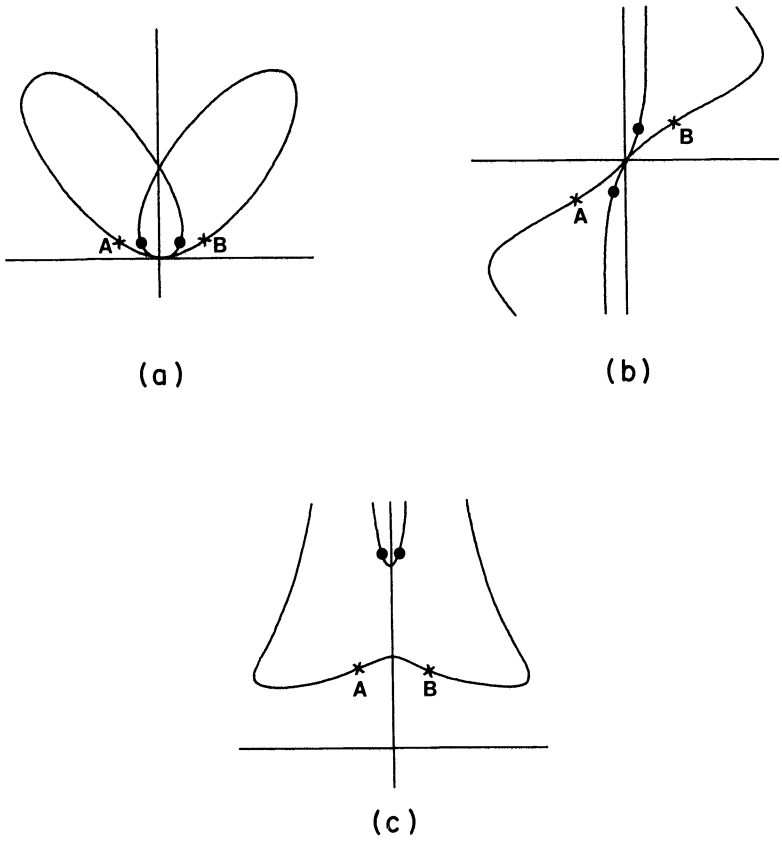


FIG. 9. (a) *The original singularity.* (b) *After one quadratic transformation.* (c) *After a second transformation: the original singularity successfully transformed into two simple points.*

To summarize, each singularity is translated to the origin and transformed into a set of nonsingular points through the application of a series of quadratic transformations. Each branch of the transformed curve intersects the exceptional line in a simple point, so this image branch can be traced from the image singularity without confusion. Therefore, upon each image branch, two points are found by tracing a short distance in either direction from the image singularity. Finally, these points are mapped back to the original curve to become refined endpoints, replacing the singularity. These new endpoints clarify the branch connectivity at the singularity and simplify the job of pairing.

The renaming of singularities has only two purposes: to clarify branch connectivity at the singularity and to simplify pairing. Since neither of these purposes depends on the exact position of the refined endpoints, the position of refined endpoints is quite flexible. In particular, we must only ensure that no convex segment is refined away. Thus, during the refinement of a convex segment $\overset{\frown}{AB}$ through the refinement of the singularity A, we must ensure that A is replaced by $A' \in \overset{\frown}{AB}$. Suppose $A \neq B$. Certainly $A' \in \overset{\frown}{AB}$ if the trace to A' remains inside the circle of radius $||A - B||/2$ centered at A. Therefore (not yet knowing B), we restrict the trace to the circle of radius $||A - E||/2$ centered at A, where $E \neq A$ is the closest endpoint to A that is not a refined endpoint associated with A.[8] Note that traces can be reversed and made arbitrarily short. Now suppose $A = B$ (i.e., $\overset{\frown}{AB}$ is a loop). In this case, A and B are associated with different images of the singularity on the image curve, say i_1 and i_2, respectively, and the image of $\overset{\frown}{AB}$ on the image curve is $\overset{\frown}{i_1 i_2}$. Certainly $A' \in \overset{\frown}{AB}$ if the image of A' is on $\overset{\frown}{i_1 i_2}$. Therefore, we restrict the trace on the image curve to the inside of the circle of radius $||i_1 - i_j||/2$ centered at i_1, where $i_j \neq i_1$ is the closest image of the singularity to i_1.

5.2. Refinement of convex segments II: Infinite segments. Convex segments with singular endpoints are not the only ones that must be refined: infinite convex segments are also problematic. The pairing process is simplified if each convex segment has two endpoints, but an infinite convex segment has only one endpoint. Therefore, a second endpoint is added to each infinite segment, as follows. Once again, this is only a renaming procedure.

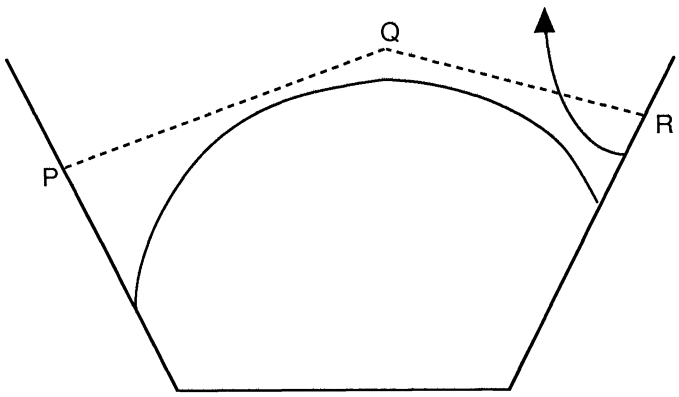


FIG. 10. $\overline{PQ}$ *and* $\overline{QR}$ *are artificial walls.*

We shall artificially bound each unbounded cell by a collection of line segments, called *artificial* (*cell*) *walls* (Fig. 10). These artificial walls are chosen carefully so

[8] The consideration of refined endpoints associated with A forces A' artificially close to A. Note that B is not a refined endpoint associated with A, since $\overset{\frown}{AB}$ existed before the refinement of A began.

that they only intersect infinite convex segments (if any) in the cell, and each of these exactly once (unless the infinite segment does not touch the original cell boundary at all and thus proceeds to infinity at both ends, in which case two intersections are allowed). The walls should also be chosen so that the resulting artificially bounded cell is a convex polygon. The creation of artificial walls that satisfy all of these properties is discussed in the Appendix (§ 10.1).

After every unbounded cell has been artificially bounded, a second endpoint is added to each infinite convex segment by making an (*artificial*) *endpoint* at its point of intersection with an artificial wall (Fig. 18). Subsequently, each infinite convex segment is represented by a finite convex segment with two endpoints. Once convex segments have been identified by pairing endpoints, a pair that contains an artificial endpoint will be recognized as an infinite convex segment.[9]

After refining both singularities and infinite segments, there are three types of endpoint: (a) endpoints defined by the original cell decomposition, called original endpoints: those that are nonsingular are still endpoints of refined segments; (b) endpoints refined from singularities, called refined endpoints, and (c) endpoints added to infinite segments, called artificial endpoints. From now on, when we speak of all of the endpoints of the decomposition, we are referring only to the collection of nonsingular original, refined, and artificial endpoints, unless otherwise stipulated. Singularities are no longer considered to be endpoints. After refinement, endpoints and cells assume the following normal form:

(i) Every endpoint has exactly two partners,

(ii) Every cell is a bounded polygon.

Refinement will not only make the pairing of endpoints easier. It will also create a cleaner set of convex segments that better reflects the curve. In particular, due to the endpoint normal form, the pairing of endpoints will create a collection of convex segments that can be ordered very easily.

5.3. Pairing of endpoints I: Properties of the partner. We are now ready to show how to pair the endpoints of convex segments. Consider a convex segment in cell C and an endpoint E of this segment. E's partner in C must obviously be another endpoint in C. Therefore, the determination of partners in all single-segment cells is trivial. Corollary 5.4 will present other conditions that E's partner must satisfy and Theorem 5.5 will show how to isolate the partner if several endpoints satisfy all of these conditions. In preparation, some terminology must be introduced and a crucial lemma proved.

Definition. Let P be a point on a curve F, and let C be the cell (or one of the cells) of F's cell partition that contains P (Figs. 11 and 12). If P is a singularity or flex, then the *inside of P's tangent* with respect to (w.r.t.) C is the halfplane bounded by P's tangent that contains C (which is unambiguous, because P's tangent defines a wall of C). If P is neither a singularity nor a flex, the inside of P's tangent is the halfplane bounded by P's tangent that contains all of the curve in the immediate neighborhood of P.[10] The inside includes the tangent, while the strict inside does not.

Let P be a flex that lies on the wall W of cell C, and let P_ϵ be a point of the

[9] A pair that contains two artificial endpoints will be recognized as an entire connected component that does not cross any of the singularity/flex tangents. See the discussion of nude components in § 6.1.

[10] The inside of P's tangent can be determined by tracing from P. The trace is restricted to C to guarantee that it stays on the same side of the tangent.

FIG. 11. *The inside of P's tangent.*

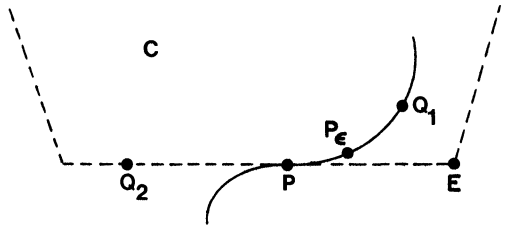


FIG. 12. *P faces both Q_1 and Q_2 with respect to C.*

curve inside C at distance $\epsilon > 0$ from P. (P_ϵ may be found by tracing the curve into C from P.) The *outside wallpoint* of W w.r.t. C is the endpoint of W that lies outside of P_ϵ's tangent, for ϵ small (E in Fig. 12).

Let P and Q be points on a curve. If P is not a flex, then P *faces* Q if Q lies on the inside of P's tangent (Fig. 11). If P is a flex lying on a wall of cell C, then P faces Q w.r.t. C if (1) Q lies on the strict inside of P's tangent w.r.t. C or (2) Q lies on P's tangent and on the opposite side of P from the outside wallpoint of P's wall w.r.t. C (Fig. 12).

Notation. $\#\{S\}$ is the number of elements in the set S and $\overline{xy}$ is the line segment strictly between x and y. It is important to note that $\overline{xy}$ does not include its endpoints x and y.

The following lemma captures the fact that, if $\overline{XY}$ is a line segment satisfying some simple properties, the intersections of the curve with $\overline{XY}$ must pair up into couples that face each other.

LEMMA 5.3. *Consider the cell partition of a curve F. Let X and Y be two nonsingular points of a convex segment in the cell C. Then*

(1) *The curve crosses*[11] $\overline{XY}$ *at an even number of points, ignoring singularities.*

(2) $\#\{P \in \overline{XY} \cap F : P \text{ faces } X \text{ w.r.t. } C\} = \#\{P \in \overline{XY} \cap F : P \text{ faces } Y \text{ w.r.t. } C\}$.

(3) *For all* $\alpha \in \overline{XY}$,

$$\#\{P \in \overline{X\alpha} \cap F : P \text{ faces } X \text{ w.r.t. } C\} \leq \#\{P \in \overline{X\alpha} \cap F : P \text{ faces } Y \text{ w.r.t. } C\}.$$

[11] If P is a point of intersection of the curve with $\overline{XY}$, then the curve *crosses* $\overline{XY}$ at P if it lies on both sides of $\overline{XY}$ in any neighborhood of P; otherwise it only touches $\overline{XY}$ at P.

Example 5.2. Fig. 13 is a hypothetical example for Lemma 5.3. The curve F crosses $\overline{XY}$ an even number of times. $\{P \in \overline{XY} \cap F : P \text{ faces } X\} = \{P_2, P_5, P_6\}$ is of the same size as $\{P \in \overline{XY} \cap F : P \text{ faces } Y\} = \{P_1, P_3, P_4\}$. Moreover, $\{P \in \overline{X\alpha} \cap F : P \text{ faces } X\} = \{P_2\}$ is smaller than $\{P \in \overline{X\alpha} \cap F : P \text{ faces } Y\} = \{P_1, P_3, P_4\}$.

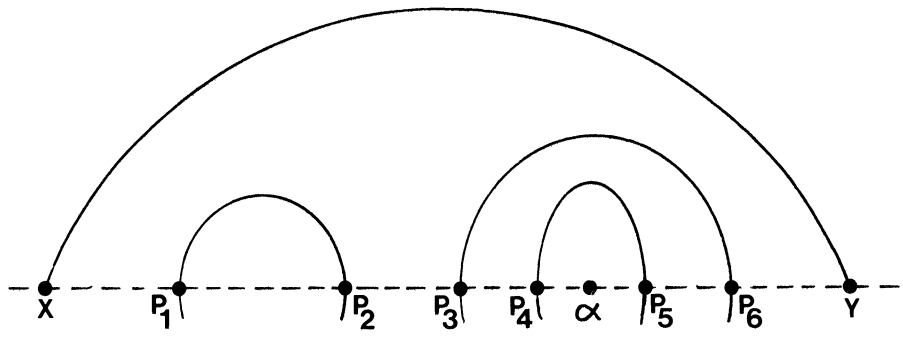


FIG. 13. *An illustration of Lemma* 5.3.

Condition (3) of Lemma 5.3 may look ominous, but it is not difficult to test. It is only shorthand for the fact that the number of intersections that face Y must always dominate the number that face X. In other words, we need only calculate information at the points of intersection $P \in \overline{XY} \cap F$, not at an infinite number of $\alpha \in \overline{XY}$.

Proof of Lemma 5.3. Consider the closed region R_{XY} bounded by $\overline{XY}$ and $\overset{\frown}{XY}$. Since $\overset{\frown}{XY}$ lies in the cell C and C is a convex polygon, $\overline{XY}$ must also lie in C. Therefore, again by convexity, R_{XY} must lie in C. Since X and Y are nonsingular and the rest of $\overset{\frown}{XY}$ lies in the interior of the cell, $\overset{\frown}{XY}$ does not contain a singularity. Therefore, the curve can only cross into R_{XY} through $\overline{XY}$. If the curve enters R_{XY}, then it must also leave, since an infinite segment cannot remain within a bounded region and an algebraic curve of finite length is closed (viz., the curve cannot stop short in the middle of R_{XY}). We claim that the point of departure D must be distinct from the point of entry E, unless all of the tangents at $D = E$ are $\overleftrightarrow{XY}$, as in Fig. 14. In particular, if $D = E$ and some tangent at D is not $\overleftrightarrow{XY}$, then at least one of the tangents of the singularity D will cross into R_{XY} and form a wall of the cell partition which will split R_{XY} in two, contradicting the fact that all of R_{XY} lies in the same cell. Therefore, with the exception of the special singularities of Fig. 14, the crossings of $\overline{XY}$ by the curve occur in pairs, called *couples*. This establishes condition (1) of the lemma.

Consider condition (2). The special singularities of Fig. 14 (as well as the points where the curve only touches $\overline{XY}$) can be ignored during the consideration of conditions (2) and (3), since they face both X and Y and contribute the same amount to the left- and right-hand sides of the expressions of conditions (2) and (3). Therefore, we can concentrate on the remaining crossings of $\overline{XY}$: the distinct couples. Let $A, B \in \overline{XY}$ be a couple and assume, without loss of generality, that A lies closer to X than B does. $\overset{\frown}{AB}$ is a convex segment since it lies within a cell of the cell partition. Therefore, A and B face each other (w.r.t. cell C). Since A faces B, A faces Y.

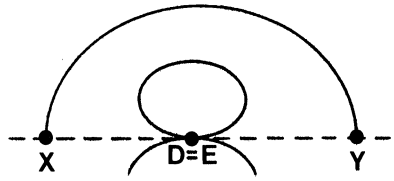


FIG. 14. *The only type of singularity that can lie on* $\overline{XY}$.

Similarly, since B faces A, B faces X. Therefore, one member of each couple faces X and the other faces Y, yielding condition (2). Moreover, the point of a couple that faces Y (A) is closer to X than the point that faces X (B), yielding condition (3). □

COROLLARY 5.4. *Let* W_1 *be an endpoint of a convex segment that lies in the cell* C *of the cell partition of a curve* F. W_1*'s partner* W_2 *in* C *must satisfy the following properties:*

(1) W_1 *and* W_2 *face each other (w.r.t.* C*).*

(2) $\#\{P \in \overline{W_1W_2} \cap F : P \text{ nonsingular, } F \text{ crosses } \overline{W_1W_2} \text{ at } P\} = 2k,\ k \in \mathbb{Z}$.

(3) $\#\{P \in \overline{W_1W_2} \cap F : P \text{ faces } W_1 \text{ (w.r.t. } C)\} =$
$\#\{P \in \overline{W_1W_2} \cap F : P \text{ faces } W_2 \text{ (w.r.t. } C)\}$.

(4) *For all* $\alpha \in \overline{W_1W_2}$,
$\#\{P \in \overline{W_1\alpha} \cap F : P \text{ faces } W_1\} \leq \#\{P \in \overline{W_1\alpha} \cap F : P \text{ faces } W_2\}$.

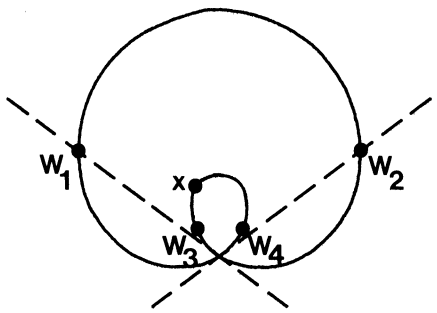


FIG. 15. W_2 *must be* W_1*'s partner.*

The conditions of Corollary 5.4 will often isolate the partner.

Example 5.3. Consider the cell partition of Fig. 15 and the cell containing the convex segments $W_1\widehat{W_2}$ and $W_3\widehat{W_4}$. Suppose that we wish to find the partner of W_1. W_3 violates condition (1) and W_4 violates condition (2), so W_2 must be W_1's partner.

We add two practical notes to Corollary 5.4. In the sequel, there will be many situations in which the conditions of Corollary 5.4 must be checked. During this computation, it is useful to keep in mind that the exact location of the refined endpoints

is not important (as discussed in the last paragraph of §5.1). This can be used to improve the ease or robustness of pathological computations. For example, if many refined endpoints are clustered close together about a singularity, it may be useful to spread these endpoints out.

The second note also concerns the checking of conditions in Corollary 5.4. Conditions (3) and (4) remain true if $\#\{P \in \overline{W_1W_2} \cap F : \text{<cond>}\}$ is replaced by $\#\{P \in \overline{W_1W_2} \cap F : \text{<cond>}, P \text{ nonsingular}, P \text{ is a crossing}\}$. Moreover, the restriction to nonsingular crossings does not change any of the proofs that use this corollary. For conciseness, we do not use the restriction. However, for practical reasons it is advisable to use it, because of the large potential for error in computing the direction that singularities on $\overline{W_1W_2}$, as well as points that touch $\overline{W_1W_2}$, face. (In theory, these singularities and touching points face both W_1 and W_2.)

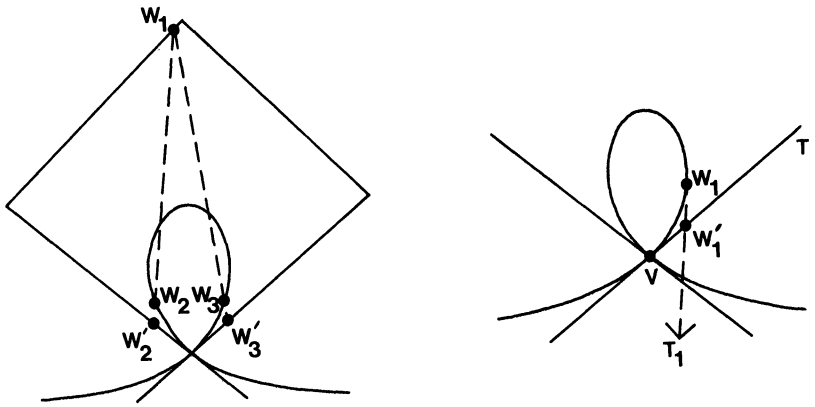


FIG. 16. *Sending refined points to the boundary.*

5.4. Pairing of endpoints II: Distinguishing between candidates. The remaining question in endpoint-pairing is how to find the partner of an endpoint W_1 in C if several endpoints in C satisfy all of the conditions of Corollary 5.4. This will be done by sorting the candidates about the cell boundary (Theorem 5.5). Unfortunately, the refinement of singularities moved some of the endpoints of convex segments into the interior of cells. Therefore, in order to allow sorting about the boundary, we must associate a point W' on the cell boundary with each refined endpoint W, as follows (Fig. 16). If $W \neq W_1$, then W' is the intersection of the ray $\overrightarrow{W_1W}$ with the cell boundary. If $W = W_1$, then W' is one of the two intersections of W_1's tangent with the cell boundary: the one that lies on a tangent of the singularity from which W_1 was derived. For notational consistency, $W' = W$ if W is not a refined endpoint.

The following theorem shows how to find the partner of W_1 when Corollary 5.4 cannot.

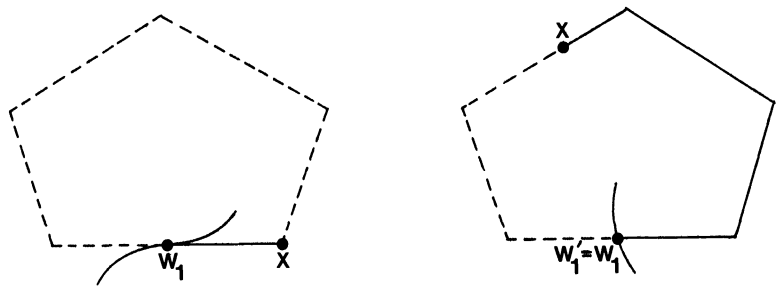


FIG. 17. *Partitioning the boundary in two halves.*

THEOREM 5.5. *Let W_1 be an endpoint in cell C of the cell partition of a curve F, $R(W_1)$ the set of endpoints in C that satisfy the conditions of Corollary* 5.4, *and $S(W_1)$ the set of endpoints in $R(W_1)$ that lie* strictly *inside of W_1's tangent. There are two cases to consider in finding the partner of W_1.*

(a) *Suppose that $S(W_1) \neq \emptyset$. Let $S'(W_1) := \{ W' : W \in S(W_1) \}$ and, if W_1 is not a flex, let $X \neq W_1'$ be the other intersection of W_1's tangent with the cell boundary, otherwise let X be the outside wallpoint of W_1's wall w.r.t. C (Fig.* 17). *W_1' and X split the cell boundary into two halves. Since every endpoint in $S'(W_1)$ will lie on the same half of the boundary, a sort of $S'(W_1)$ from W_1' to X is well defined. Let $S_1', S_2', \cdots, S_p'$ be the result of this sort (i.e., S_i' is encountered before S_{i+1}' in a traversal of the cell boundary from W_1' to X). The partner of W_1 in C is S_p (the endpoint associated with S_p').*

(b) *Suppose that $S(W_1) = \emptyset$, and let $T(W_1)$ be the set of endpoints in $R(W_1)$ that lie on the same wall as W_1. The partner of W_1 in C is the element of $T(W_1)$ that is closest to W_1.*

Example 5.4. Consider the computation of W_1's partner in Fig. 18, where W_1 is the endpoint of an infinite convex segment. $R(W_1) = S(W_1) = \{W_2, W_3, W_4\}$ and $S'(W_1) = \{W_2, W_3, W_4'\}$. The sorted order of $S'(W_1)$ along the boundary from W_1' to X is W_3, W_4', W_2, so W_2 is the partner of W_1. Since W_2 is an artificial endpoint, W_1 must be the endpoint of an infinite convex segment.

Consider the computation of the partner of W_1 in Fig. 19, where $S(W_1) = \emptyset$. V_1, V_2 and V_4 are ruled out by condition (1) of $R(W_1)$, while V_3 and V_6 are ruled out by condition (2). Therefore, $T(W_1) = \{V_5, W_2\}$. W_2 is the closest element of $T(W_1)$ to W_1, so it is W_1's partner.

Proof of Theorem 5.5. A key lemma in this proof (as well as others) is Lemma 10.2 of the Appendix. The reader is urged to refer to this lemma and its proof when it is used below.

(a) Suppose that $S(W_1) \neq \emptyset$. Let W_2 be W_1's partner, and let $\widehat{W_1W_2}$ be the boundary of the cell from W_1' to W_2', such that $X \notin \widehat{W_1W_2}$ (Fig. 20). It is sufficient to show that (i) $W_2' \in S'(W_1)$ and (ii) $S'(W_1) \subset \widehat{W_1W_2}$. Indeed, suppose that $W_2' \in S'(W_1) \subset \widehat{W_1W_2}$ and consider a traversal of the cell boundary from W_1' to X. Since W_1' and W_2' are extreme points of $\widehat{W_1W_2}$ and $X \notin \widehat{W_1W_2}$ (by definition of $\widehat{W_1W_2}$), W_2' is the last element of $S'(W_1)$ that is met during this traversal. In other

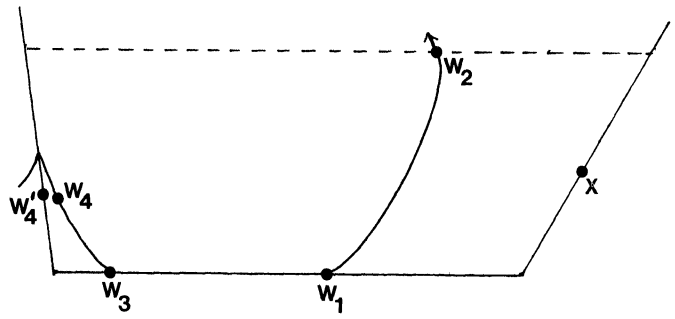


FIG. 18. *The partner of W_1 is W_2.*

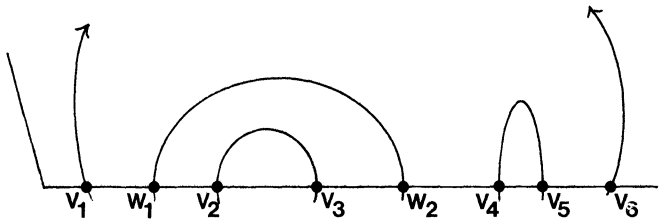


FIG. 19. *The partner of W_1 is again W_2.*

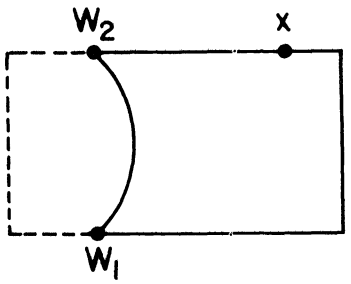


FIG. 20. $\widehat{W_1W_2}$ *is dotted.*

words, $W_2' = S_p'$ ($W_2 = S_p$) as desired. (There is no ambiguity in choosing the last member of $S'(W_1)$ or in associating S_i' with S_i, since it can be shown that $S_i' \neq S_j'$ whenever $i \neq j$ [18, p. 75].)

We will first show that (ii) $S'(W_1) \subset \widehat{W_1W_2}$. Let $s \in S(W_1)$. By Lemma 10.2 of the Appendix, $\vec{W_1s}$ does not cross $\widehat{W_1W_2} \setminus \{W_2\}$. In particular, $\overline{W_1s'}$ does not cross $\widehat{W_1W_2} \setminus \{W_2\}$. Therefore, s' must either lie outside of W_1's tangent or on $\widehat{W_1W_2}$ (Fig. 20). Since s, as a member of $S(W_1)$, lies on the strict inside of W_1's tangent, so must s'. Therefore, $s' \in \widehat{W_1W_2}$ and $S'(W_1) \subset \widehat{W_1W_2}$, as desired.

We now show that $W_2 \in S(W_1)$ (which implies (i) $W_2' \in S'(W_1)$). $W_2 \in R(W_1)$ by Corollary 5.4, so it suffices to show that W_2 lies strictly inside of W_1's tangent. Since $\widehat{W_1W_2}$ is a convex segment, W_2 lies on the inside of W_1's tangent, and we must only show that W_2 does not lie on W_1's tangent. Suppose that W_2 lies on W_1's tangent. Since $\overline{W_1W_2}$ is a subsegment of W_1's tangent, $S(W_1) \cap \overline{W_1W_2} = \emptyset$. Since $S'(W_1)$ is found by following rays from W_1 through elements of $S(W_1)$, we also have $S'(W_1) \cap \overline{W_1W_2} = \emptyset$. Finally, since $\overline{W_1W_2}$ is part of a cell wall (Lemma 10.4 of the Appendix), we have $\widehat{W_1W_2} = \overline{W_1W_2}$. The combination of $S'(W_1) \cap \overline{W_1W_2} = \emptyset$ and $S'(W_1) \subset \widehat{W_1W_2} = \overline{W_1W_2}$ forces $S'(W_1) = \emptyset$, which contradicts our initial assumption. Thus, W_2 does not lie on W_1's tangent and $W_2 \in S(W_1)$.

(b) The statement of the theorem has been verified if $S(W_1) \neq \emptyset$. Now suppose that $S(W_1) = \emptyset$. We first show that $T(W_1)$ is well defined and contains W_2. If W_1 does not lie on a cell wall, then $W_2 \in S(W_1)$: $W_2 \in R(W_1)$ (as W_1's partner); W_2 does not lie on W_1's tangent (Lemma 10.4); and W_2 lies inside W_1's tangent (because $\widehat{W_1W_2}$ is convex). Thus, W_1 must lie on a cell wall and $T(W_1)$ is well defined. If W_2 lies strictly inside W_1's wall (w.r.t. C), it also lies strictly inside W_1's tangent (Lemma 10.4). That is, if $W_2 \notin T(W_1)$, then $W_2 \in S(W_1)$. Therefore, $W_2 \in T(W_1)$.

Suppose that W_2 is not the closest member of $T(W_1)$ to W_1, and let $U \neq W_2$ be the closest. Since W_1 faces U, U must lie on $\overline{W_1W_2}$. Using a familiar argument from Lemma 5.3, the nonsingular points of entry and departure of the curve into the closed region bounded by $\overline{W_1W_2}$ and $\widehat{W_1W_2}$ must pair up along $\overline{W_1W_2}$, such that each pair defines a convex segment. In particular, each nonsingular point of the curve on $\overline{W_1U}$ that faces W_1 must pair with a nonsingular point on $\overline{W_1U}$ that faces W_2. But, since $U \in R(W_1)$, the number of nonsingular points on $\overline{W_1U}$ that face W_2 is exactly equal to the number of nonsingular points on $\overline{W_1U}$ that face W_1 (Corollary 5.4). Thus, each nonsingular point on $\overline{W_1U}$ that faces W_2 is paired with some nonsingular point on $\overline{W_1U}$. In particular, no nonsingular point on $\overline{W_1U}$ that faces W_2 is paired with a point on $\{U\} \cup \overline{UW_2}$. But this leads to a contradiction: U is nonsingular (since it is an endpoint after the refinement stage) and it faces W_1 (since $U \in R(W_1)$), so it should pair with a nonsingular point of the curve on $\overline{W_1U}$ that faces W_2. We conclude that W_1's partner W_2 must be the closest element of $T(W_1)$ to W_1, completing the proof of this pairing theorem. □

A plane algebraic curve can now be properly decomposed into convex segments, since the endpoints of the convex segments can be properly paired together. Note that it is now simple to sort the convex segments. The endpoints form a doubly linked list that defines the order of the convex segments and makes it easy to traverse the curve, convex segment by convex segment.

5.5. Computation of singularities and flexes. The above convex decomposition of an algebraic curve requires the singularities and flexes of the curve, as well

as their tangents. The singularities of a curve $f(x, y) = 0$ are the solution set of the system $\{f_x = 0, f_y = 0, f = 0\}$, while the points of inflection are the nonsingular intersections of the curve with its Hessian (the determinant of the matrix of double derivatives of the curve's equation) [28]. These systems of polynomial equations can be solved efficiently for all of their point solutions by using U-resultant schemes [10]. Furthermore, using bounds on the minimum gap between roots of a polynomial, and bounds on the size of the coefficients and degree of the original curve, we can straightforwardly estimate the bits of accuracy required in computing the solution to ensure that the error in the value of the curve's derivatives is within the gap. This in turn ensures the correctness of the quadratic transformations applied to the curve and centered at the singularity.

The restriction of points of inflection to flexes (see the second paragraph of § 5) is straightforward [18, p. 44]. The tangents of a singularity of the curve $f = 0$ can be found by translating the singularity to the origin. The equations of the tangents are the factors of the translated f's order form (the polynomial consisting of the terms of lowest degree) [28]. Finally, after the curve has been translated to projective space by homogenizing its equation to $f(x, y, z) = 0$ (where z is the homogenizing variable), the tangent of a flex P is $f_x(P)x + f_y(P)y + f_z(P)z = 0$ [28]. This completes our description of the convex decomposition of an algebraic curve.

6. Point location. The second key problem in the convex segment method of sorting is point location in a convex decomposition: given a point, we must identify the convex segment that contains it. This is an extension to the curved domain of the well-known problem of point location in a planar subdivision. We will show how to locate a point on a convex segment (§ 6.1), a general curve segment (§ 6.2), and a connected component of a curve (§ 6.3).

6.1. Point location I: On a convex segment. A decomposition is not very useful unless it is possible to locate points in it. In the case of sorting, point location is necessary in order to partition a set of points into convex segments. Since a convex segment is identified by its endpoints, finding the convex segment that contains a point is equivalent to finding the endpoints that bound this convex segment. This problem is analogous to finding the partners of a given endpoint (§ 5.4), since both problems are instances of the more general question: "What are the two endpoints associated with a given point?" We continue to use refined segments for point location, because it is important for an endpoint in a cell C to uniquely identify a convex segment in C. However, recall that points that do not lie on any refined segment are still located on a convex segment: if $x \in \overset{\frown}{AB} \setminus \overset{\frown}{A'B'}$, where $\overset{\frown}{A'B'}$ is the refined segment associated with the original segment $\overset{\frown}{AB}$, then x is located on $\overset{\frown}{A'B'}$.

It is easy to locate a point in the proper cell, using well-known algorithms for point location in a planar subdivision [20], [24]. Artificial walls are ignored during this step: a point is considered to lie in an artificially bounded cell C as long as it lies in the unbounded cell associated with C. If, as is often the case, a point lies in a cell with only one convex segment, then it is obvious to which convex segment it belongs. Otherwise, Theorem 6.1 and Lemma 6.2 can be used to locate a point on the proper convex segment. Lemma 6.2 shows how to locate points on a special type of convex segment, a nude component. Theorem 6.1 shows how to locate points on other convex segments. Before we get to these results, we must define a nude component and the type of a convex segment.

Definition. A *connected component* of a curve is a maximal subset of the curve

such that there exists a continuous path on the curve between any two points of the subset.

For example, a hyperbola has two connected components. A connected component that lies entirely inside of a cell, intersecting none of the walls (including artificial walls) of the cell partition, is called a *nude component* (Fig. 21). It is nude because, unlike other connected components, it does not contain any endpoints of convex segments. This lack of endpoints makes nude components a special case for point location. A nude component is convex, since it does not contain any singularities or flexes.

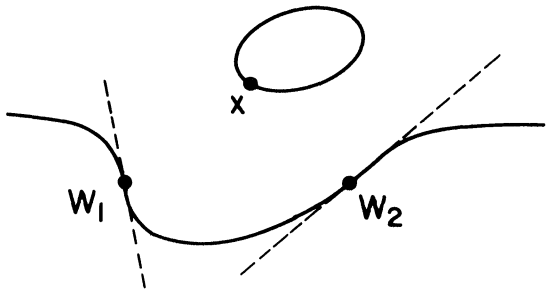


FIG. 21. *x lies on a nude component.*

Definition. Let $\overset{\frown}{P\infty}$ be an infinite convex segment before refinement, $x \in \overset{\frown}{P\infty}$. x's tangent defines two directions: the *finite (respectively, infinite) direction from* x is the direction along x's tangent that the curve leaves x in traveling to the endpoint P (respectively, to infinity). Let R be the ray along x's tangent in the infinite direction from x. The *type of an infinite convex segment* is clockwise-convex or counterclockwise-convex. $\overset{\frown}{P\infty}$ is *clockwise-convex* if and only if R enters the halfplane defined by the inside of x's tangent as it rotates clockwise (Fig. 22). (Since the clockwise direction of rotation depends on one's perspective, a perspective from a fixed side of the plane containing the curve must be maintained for all type computations.)

The type of an infinite convex segment is well defined, because it is independent of x. The computation of this type is straightforward. The only nontrivial step is the computation of the infinite direction from a point, which is discussed in Lemma 10.5 of the Appendix.

We are finally ready to show how to locate a point on the proper convex segment. There are three main cases to this point location: (i) the point lies inside the bounded cell and on a refined segment, (ii) the point lies inside the bounded cell and not on a refined segment, and (iii) the point lies outside the bounded cell.

THEOREM 6.1. *Let x be a point of curve F that lies in cell C.*[12] *If x is an endpoint, it is located on the refined segment in C associated with this endpoint. If x is a singularity, it is located on one and only one of the refined segments in C associated with this singularity.*

[12] Recall that, if C is an artificially bounded cell, a point is considered to lie in C as long as it lies in the unbounded cell associated with C.

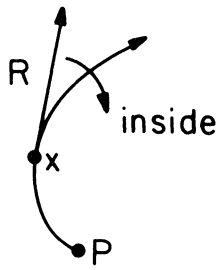


FIG. 22. *A clockwise-convex segment.*

Otherwise, let $S(x) = \{$*endpoints* W *in* C |
(1) x *lies on the strict inside of* W*'s tangent,*
(2) W *lies on the strict inside of* x*'s tangent,*
(3) $\#\{P \in \overline{xW} \cap F : P \text{ faces } x\} = \#\{P \in \overline{xW} \cap F : P \text{ faces } W\}$,
(4) *For all* $\alpha \in \overline{xW}$, $\#\{P \in \overline{x\alpha} \cap F : P \text{ faces } x\} \leq \#\{P \in \overline{x\alpha} \cap F : P \text{ faces } W\}\}$, *and let* $S''(x) = \{\ W'' : W \in S(x)\ \}$, *where* W'' *is the intersection of* $x\vec{W}$ *with the boundary of* C. ($S(x)$ *should be computed by the method described in* § 6.1.1.) *If* $S(x) = \emptyset$, *then* x *lies on a nude connected component and Lemma* 6.2 *is used to locate* x *on the correct nude component.*

(i) *Suppose that* x *lies inside the bounded cell* C. *Let* x_1 *and* x_2 *be the two points of intersection of* x*'s tangent with the boundary of* C, *and let* $S_1'', S_2'', \cdots, S_p''$ *be the result of a sort of* $S''(x)$ *from* x_1 *to* x_2 *along the cell boundary.* x *either lies on* S_1*'s or* S_p*'s convex segment, and there are three cases to consider:* (a) *if* S_1 *and* S_p *are partners,*[13] x *lies on* $\overset{\frown}{S_1 S_p}$; (b) *otherwise, if* S_1 *is a refined endpoint and* S_p *is not,* x *lies on* S_1*'s convex segment;* (c) *if both* S_1 *and* S_p *are refined,* x *lies on* S_1*'s convex segment if and only if* S_1'' *is one of the extreme points of the sort of* $\{S_1'', S_p'', T_1, T_p\}$ *from* x_1 *to* x_2, *where* T_1 *and* T_p *are the singularities associated with* S_1 *and* S_p.

(ii) *Suppose that* x *lies outside of* C. *Let* $A(x) = \{W \in S(x) : W$ *is an artificial endpoint*$\}$ *and let* $Q \in A(x)$ *be the unique endpoint in* $A(x)$ *whose convex segment is of the same type as* x*'s segment. Then* x *lies on* Q*'s convex segment.*

Example 6.1. In Fig. 21, $S(x) = \emptyset$ and x lies on a nude component.

Consider the cell of Fig. 15 that contains the convex segments $\overset{\frown}{W_1 W_2}$ and $\overset{\frown}{W_3 W_4}$. W_1 does not satisfy condition (2) of $S(x)$ and W_2 does not satisfy condition (3). Thus, $S(x) = \{W_3, W_4\}$ and x must lie on $\overset{\frown}{W_3 W_4}$.

Consider the cell partition of Fig. 3. $S(P_1) = \{W_1, W_2, W_5, W_6\}$, which does not resolve the question of P_1's convex segment. Let x_1 and x_2 be the two points of intersection of P_1's tangent with the cell boundary. Since the sort of $S''(P_1)$ from x_1 to x_2 is (W_1, W_6, W_5, W_2), and W_1 and W_2 are partners, P_1 must lie on $\overset{\frown}{W_1 W_2}$.

In Fig. 23, the sort of $S''(x)$ from x_1 to x_2 is (W_1'', W_2, W_3, W_4''), W_1 and W_4 are not partners, and both of them are refined. Since the sort of W_1'', W_4'', T_1, and T_p from x_1 to x_2 is $\{W_1'', W_4'', T_1 = T_p\}$, we conclude that x lies on W_1's segment.

[13] We will see that S_1 and S_p are partners whenever x lies on a refined segment.

In Fig. 24, x lies outside of the bounded cell and $A(x) = \{W_1, W_3\}$. x's and W_1's segment are clockwise-convex, but W_3's segment is counterclockwise-convex. Thus, x must lie on W_1's segment.

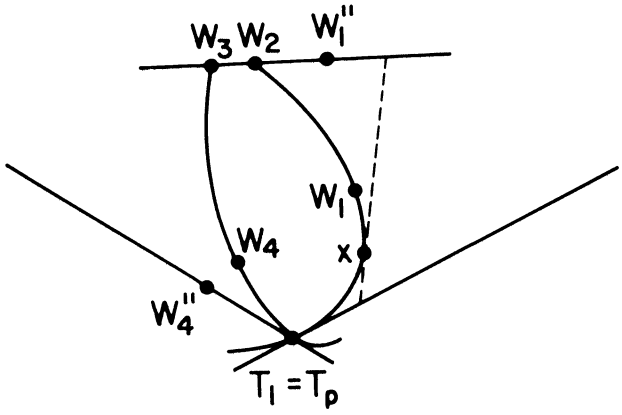


FIG. 23. *x lies on W_1's convex segment.*

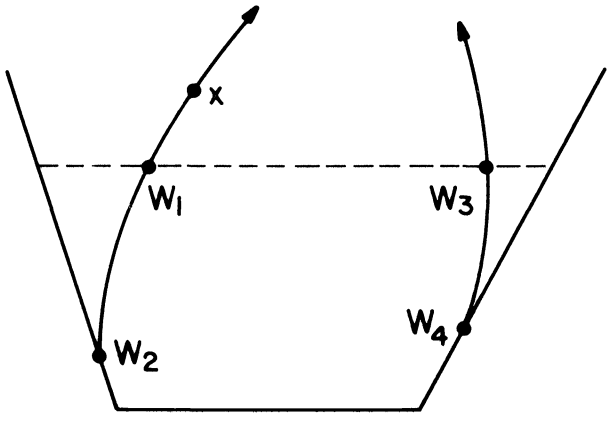


FIG. 24. *x lies on W_1's convex segment.*

Proof of Theorem 6.1. Consider the decomposition of the curve into convex segments before refinement of singularities and infinite segments. In particular, consider the convex segment SEG in this decomposition that contains x. After refinement of the convex segments, SEG is represented by a refined segment $\overset{\frown}{W_1W_2} \subset$ SEG. When $x \in \overset{\frown}{W_1W_2}$, the proof is very similar to that of Theorem 5.5. (Note that it will usually

be the case that $x \in \overset{\frown}{W_1 W_2}$. Indeed, in many cases $\overset{\frown}{W_1 W_2} = \text{SEG}$.) However, new ideas are required when $x \in \text{SEG} \setminus \overset{\frown}{W_1 W_2}$, in which case x either lies on a segment of the curve that was removed while refining a singularity, or it lies on a segment that was removed while refining an infinite convex segment. In the former case, assume without loss of generality that $x \in \overset{\frown}{W_{\text{sing}} W_1}$, where W_{sing} is the singularity that was refined to make W_1. In the latter case, assume without loss of generality that W_1 is the artificial endpoint of x's infinite segment and $x \in \overset{\frown}{W_1 \infty}$.

We first notice that $W_1, W_2 \in S(x)$:

(1)–(2) x, W_1, and W_2 lie on the same convex segment and $x \neq W_1, W_2$.

(3)–(4) Lemma 5.3 ($X = x$, $Y = W_1$ or W_2).

Thus, if $S(x) = \emptyset$, then x must lie on a nude component containing no endpoints. Suppose $S(x) \neq \emptyset$. Notice that a sort of $S''(x)$ from x_1 to x_2 along the cell boundary is well defined, since all of $S''(x)$ lies on the same side of x's tangent (condition (2) of $S(x)$).

(i) Suppose that x lies inside C. Thus, $x \in \overset{\frown}{W_1 W_2}$ or $x \in \overset{\frown}{W_{\text{sing}} W_1}$. In either case, $W_1{}''$ is either the first or last element of the sort of $S''(x)$ from x_1 to x_2, by Corollary 10.3 of the Appendix. That is, $W_1{}'' = S_1''$ or S_p''; or $W_1 = S_1$ or S_p, since $S_i'' = S_j''$ if and only if $S_i = S_j$ [18, p. 75]. We conclude that x lies on S_1's convex segment or S_p's convex segment. We must decide which one.

If S_1 and S_p are partners, then S_1's convex segment is the same as S_p's convex segment, and the decision is easy. Notice that this is the case whenever $x \in \overset{\frown}{W_1 W_2}$. Indeed, if $x \in \overset{\frown}{W_1 W_2}$, Corollary 10.3 can be applied to W_2 just as it was to W_1, yielding $W_2{}'' = S_1''$ or S_p''. That is, $\{W_1, W_2\} = \{S_1, S_p\}$, and S_1 and S_p are partners.

Assume that S_1 and S_p are not partners. Thus, $x \in \overset{\frown}{W_{\text{sing}} W_1}$ ($x \notin \overset{\frown}{W_1 W_2}$) and W_1 is a refined endpoint (i.e., an endpoint derived from a singularity). If S_1 is refined and S_p is not, then it is clear that $W_1 = S_1$ and that x lies on S_1's convex segment.

Thus, further assume that both S_1 and S_p are refined and x lies on S_1's segment, $\overset{\frown}{S_1 T_1}$. We shall establish that S_1'' is, while S_p'' is not, an extreme point in the sort of $\{S_1'', S_p'', T_1, T_p\}$ from x_1 to x_2. This will complete the proof of case (i). First note that S_p'' is not an extreme point in this sort: S_1'' is closer to x's tangent than S_p'', because $\vec{xs}$ does not cross $\overset{\frown}{xS_1} \setminus \{S_1\}$ for any $s \in S(x)$ (Lemma 10.2); similarly, on the other side, T_1 is closer to x's tangent than S_p'', because $\vec{xs}$ does not cross $\overset{\frown}{xT_1} \setminus \{T_1\}$ for any $s \in S(x)$ (Lemma 10.2). Finally, note that S_1'' **is** an extreme point in the sort (Fig. 25). T_1 clearly does not lie between S_1'' and x's tangent. Suppose, for the sake of contradiction, that T_p lies between S_1'' and x's tangent. The original convex segment that contains x divides the cell into two regions: let R_{out} be the region that contains the outside of x's tangent. The entire segment $\overset{\frown}{S_p T_p}$ must lie in R_{out}, because T_p does and $\overset{\frown}{S_p T_p}$ cannot cross x's segment without making a singularity. However, $S_p \in R_{\text{out}}$ implies that S_p lies outside x's tangent or x's segment crosses $\overline{xS_p}$, both of which contradict $S_p \in S(x)$ (using condition 2 of $S(x)$ and Lemma 10.1, respectively). Therefore, we conclude that S_p'', T_1, and T_p do not lie between S_1'' and x's tangent, forcing S_1'' to be an extreme point in the sort of $\{S_1'', S_p'', T_1, T_p\}$.

(ii) Suppose that x lies outside of C. That is, x lies on the infinite segment $\overset{\frown}{W_1 \infty}$, according to the assumptions at the beginning of the proof. We wish to show that we can locate x on W_1's convex segment by finding an artificial endpoint in $A(x)$ whose

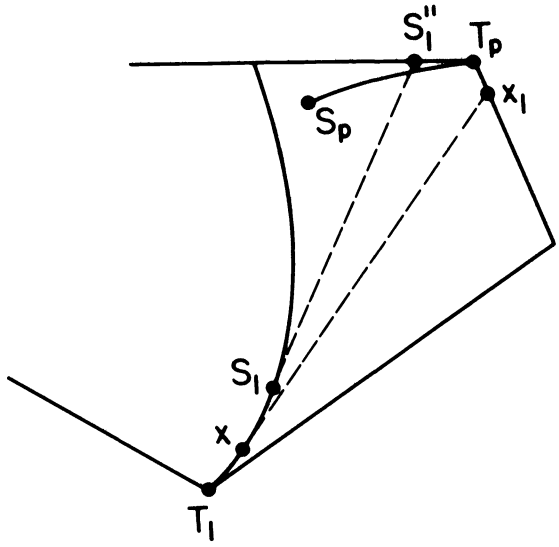


FIG. 25. *An illegal position for* T_p.

convex segment is of the same type as x's segment. That is, we wish to show that $\{a \in A(x) : a$'s convex segment is of the same type as x's$\} = \{W_1\}$. One direction is simple: $W_1 \in A(x)$ since we have shown that $W_1 \in S(x)$, and W_1's convex segment is certainly of the same type as x's, because it is the same segment. For the other direction, let $a \in A(x)$ such that a's convex segment is of the same type as x's. The idea of the proof is to show that a's convex segment enters a bounded region and cannot legally leave except through x. This will prove $x \in \widehat{a\infty}$, implying $a = W_1$ as desired.

The first step of the proof is to find a region that contains $\widehat{x\infty}$ and $\widehat{a\infty}$, where $\widehat{x\infty}$ (respectively, $\widehat{a\infty}$) is the part of x's (respectively, a's) segment starting at x (respectively, a) and proceeding to infinity. Note that $\widehat{x\infty}$ and $\widehat{a\infty}$ lie outside of the artificially bounded cell. We shall show that a containing region for $\widehat{x\infty}$ and $\widehat{a\infty}$ is R_{xa}, the intersection of the inside of x's tangent and the inside of a's tangent (Fig. 26). R_{xa} is nonempty because x and $a \in A(x) \subset S(x)$ face each other. We shall only show $\widehat{x\infty} \subset R_{xa}$, since the proof of $\widehat{a\infty} \subset R_{xa}$ is entirely analogous, $\widehat{x\infty}$ cannot leave R_{xa} through x's tangent, by the convexity of $\widehat{x\infty}$. Suppose that $\widehat{x\infty}$ leaves R_{xa} through y on a's tangent (Fig. 27). We shall establish a contradiction, thus proving that $\widehat{x\infty}$ is restricted to R_{xa}. Let R_{xya} be the region bounded by $\overline{xa}$, $\widehat{xy}$, and a's tangent. a's segment cannot pass through $\widehat{xy}$ (it would cause a singularity inside the cell), a's tangent (by the convexity of a's segment), or $\overline{ax}$ (Lemma 10.1), and it cannot double back through a (this would cause a singularity inside the cell again). Thus, when a's segment enters R_{xya}, it cannot leave. This leads to a contradiction. If $\widehat{a\infty}$ enters R_{xya}, we get the contradiction that an infinite, unbounded segment lies in the finite, bounded region R_{xya}. If the other half of a's segment enters R_{xya}, we get the

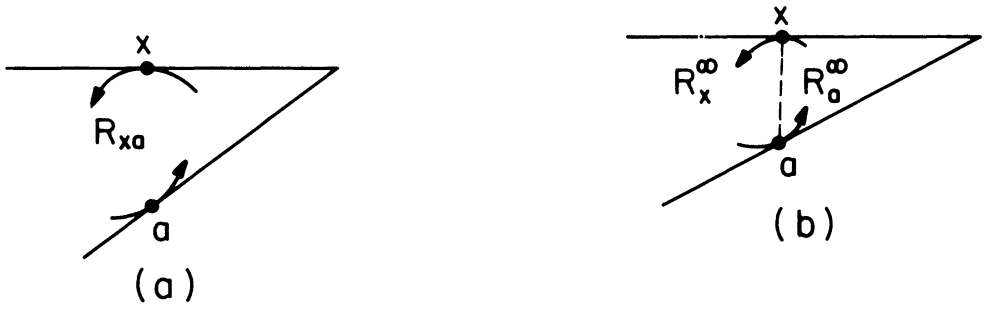


FIG. 26. *A region that contains* $\widehat{x\infty}$ *and* $\widehat{a\infty}$.

contradiction that this segment cannot reach the boundary of the cell C (R_{xya} lies in the interior of C by the convexity of C), although all convex segments originally start on a cell boundary.

We now try to restrict $\widehat{x\infty}$ and $\widehat{a\infty}$ to even smaller regions. R_{xa} is split into two subregions by the line segment $\overline{xa}$: let R_x^∞ (respectively, R_a^∞) be the subregion that contains the infinite direction from x (respectively, a), as shown in Fig. 26(b). $\widehat{x\infty}$ is restricted to R_x^∞: it starts out in R_x^∞, it cannot intersect $\overline{xa}$ (Lemma 10.1), it cannot pass through x again (since this would cause a singularity inside the cell), and it cannot intersect a (since the unique intersection of x's convex segment with the artificial boundary occurs before $\widehat{x\infty}$). By the same proof, $\widehat{a\infty}$ is restricted from passing out of R_a^∞, with one exception: it can escape through x. Thus, $\widehat{a\infty} \subset R_a^\infty$ unless $x \in \widehat{a\infty}$.

We shall finish the proof by showing that R_a^∞ is a finite, bounded region. This will establish that $\widehat{a\infty} \subset R_a^\infty$ is impossible (since $\widehat{a\infty}$ is an infinite segment) and imply $x \in \widehat{a\infty}$ as desired. To show that R_a^∞ is bounded, it is sufficient to show that $R_x^\infty \neq R_a^\infty$: only one of the two subregions of R_{xa} is unbounded (the two lines bounding R_{xa} are not parallel, since $\widehat{x\infty}$ could not stay both convex and infinite while remaining entirely between x's tangent and another parallel line) and R_x^∞ is

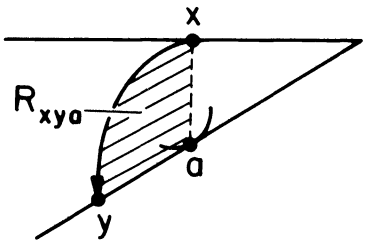


FIG. 27. $\widehat{x\infty}$ *cannot intersect* a*'s tangent.*

certainly unbounded (since it contains the infinite segment $\widehat{x\infty}$). Let V_1 (respectively, V_2) be the tangent ray from x (respectively, a) in the infinite direction, and assume without loss of generality that x's and a's convex segments (which are of the same type) are counterclockwise-convex. Thus, as V_1 and V_2 rotate counterclockwise, they both enter R_{xa}. If V_1 and V_2 point to the same side of $\overline{xa}$, then as they both rotate counterclockwise, one of them will leave R_{xa}. (Refer to Fig. 26(a).) Therefore, V_1 and V_2 must point to opposite sides of $\overline{xa}$, implying $R_x^\infty \neq R_a^\infty$.

In conclusion, since $R_a^\infty \neq R_x^\infty$ and only one of the two subregions of R_{xa} is unbounded, R_a^∞ must be bounded and $\widehat{a\infty}$ must escape from it through x, implying $a = W_1$. Therefore, $\{a \in A(x) : a\text{'s convex segment is of the same type as } x\text{'s}\} = \{W_1\}$. □

If there is only one nude component in a cell, then Theorem 6.1 can successfully locate a point on this convex segment. However, if there is more than one nude component in the cell, then the following lemma must be used to locate points on these nude components.

LEMMA 6.2. *Let P and Q be points that lie on nude components in the same cell. P and Q lie on the same nude component if and only if Q lies in $S(P)$, where $S()$ is as in Theorem* 6.1.

Proof. Let P and Q lie on nude components M and N, respectively. If $M = N$, then P and Q lie on the same convex segment, so $Q \in S(P)$ by Lemma 5.3. Suppose that $M \neq N$. Nude components do not intersect, since they do not contain any singularities. Therefore, there are only three cases to consider: M lies inside N, N lies inside M, and neither lies inside the other. In all three cases, it is straightforward to show that Q violates one of the conditions of $S(P)$. □

6.1.1. Speeding up point location. Point location can be made faster if the set $S(x)$ of Theorem 6.1 can be computed more quickly. Indeed, this is the only computation in the theorem that could be expensive. Since the expense is concentrated in testing conditions (3) and (4) (which involve line-curve intersections, whereas conditions (1) and (2) are simple to test), we must avoid testing conditions (3) and (4). The idea is to eliminate as many endpoints as possible from $S(x)$ without using conditions (3) and (4). First, eliminate any endpoints in the cell that do not satisfy conditions (1) and (2) of $S(x)$. Next observe that if x is located on $\widehat{W_1W_2}$ (whether on the refined segment or not), then $W_1, W_2 \in S(x)$. (Refer to the proof of Theorem 6.1.) Thus, if $W \notin S(x)$, we can eliminate W's partner from $S(x)$ as well, since x cannot lie on W's segment. As many endpoints as possible should be eliminated using this observation. If only two endpoints (say V and W) remain after these two rounds of elimination, we can immediately conclude that x lies on V's segment. If more than two endpoints remain, we are forced to use conditions (3) and (4) to test their membership in $S(x)$. However, we have avoided testing these conditions on many endpoints. As above, whenever an endpoint fails condition (3) or (4), its partner should also be immediately eliminated from $S(x)$.

This completes our description of techniques that are needed for sorting by the convex segment method. We digress for a moment to show how the theory that we have developed can be used to solve two important problems (although they are not needed for sorting): locating a point on an arbitrary segment and deciding whether two points lie on the same connected component.

6.2. Point location II: On an arbitrary segment. Once it is known how to locate a point on a convex segment of a curve's convex decomposition, it is straight-

forward to solve the more general problem of locating a point on an arbitrary segment of the curve. Consider a segment $\overset{\frown}{AB}$ of curve C and a point P on C. To decide if P lies on $\overset{\frown}{AB}$, we decompose C into an ordered set of convex segments and compute the convex segments that contain P, A, and B (say C_P, C_A, and C_B, respectively). If C_P, C_A, and C_B are distinct, P lies on $\overset{\frown}{AB}$ if and only if C_P lies in between C_A and C_B. Degenerate cases occur if C_P, C_A, and C_B are not distinct, and require more subtlety. For example, if $C_P = \overset{\frown}{EF} = C_A \neq C_B$, then the decision is made by sorting P, A, E, and F along $\overset{\frown}{EF}$, using Theorem 4.1: $P \in \overset{\frown}{AB}$ if and only if the order is E, P, A, F (respectively, E, A, P, F) and $\overset{\frown}{AB}$ leaves A towards E (respectively, F). (A method for deciding whether $\overset{\frown}{AB}$ leaves A towards E or F is described in footnote 3.) Other cases are similar. In short, point location on an arbitrary segment is easily reducible to point location on a convex segment.

6.3. Curves with many connected components. It should now be clear that the convex segment method can sort points on any algebraic curve. In particular, it can sort points that are strewn over several connected components of a curve, with no more difficulty than sorting points on a single component. This is another advantage of the convex segment method over the parameterization method, because it is not clear how the latter method could deal with points on several components, even if we allow nonrational parameterizations. Would each connected component have a separate parameterization? If so, how would the single equation of a curve produce several independent parameterizations? If not, how would we determine the range of parameter values that is associated with each connected component?

A very useful test for a curve with several components is whether two points lie on the same connected component. For example, with this capability it is reasonable to define an edge of a solid model as a particular connected component of a multi-component curve, since the test allows us to restrict intersections with the curve to this connected component. The following lemma shows that our decomposition of the curve into convex segments makes it simple to decide whether two points lie on the same connected component.

LEMMA 6.3. *Let P and Q be two points of a curve. If both P and Q lie on nude components, use Lemma* 6.2. *Otherwise, let P and Q lie on convex segments $\overset{\frown}{AB}$ and $\overset{\frown}{CD}$, respectively. Define $v \equiv w$ if $\overset{\frown}{vw}$ is a convex segment or v and w are linked refined endpoints (such as V_1 and V_2 in Example* 5.1), *and extend this relation into an equivalence relation on endpoints by taking its reflexive, symmetric, and transitive closure. Then P and Q lie on the same connected component if and only if $A \equiv C$.*

Two other decompositions of an algebraic curve, Collins' cylindrical algebraic decomposition [13], [4] and Canny's stratification [10], can also be used to separate a curve into connected components and thus decide whether two points lie on the same connected component.

6.4. Broad comparison of methods. Let us compare the convex segment method of sorting with the tracing and parameterization methods. The convex segment method leaps along the curve (by convex segments) like the brute-force tracing method. However, its jumps are large while the tracing method's jumps must be very small. Moreover, once the convex segments have been computed (which can be done once and for all in a preprocessing step), each jump of the convex segment method can be done very quickly, whereas the tracing method must grope for some time (by

applying Newton's method) to find the destination of each jump. In short, the convex segment method makes large, bold jumps while the tracing method makes small, timid ones.

The convex segment method is similar to the parameterization method because they both reduce the sorting problem to an easier one. The parameterization method observes that the sorting of points on a line is simple and tries to unwind the curve into a line by parameterizing it. Rather than trying to reduce the entire problem, the convex segment method divides the problem up into many smaller ones (viz., the sorting of points on a convex segment). We shall see that the many small reductions of the convex segment method can be done more quickly than the single, large reduction of the parameterization method.

The convex segment method incorporates preprocessing, since the convex decomposition of a curve can be precomputed. As a result, the actual sorting is usually very efficient. We might consider the parameterization of a curve to be preprocessing, but the subsequent runtime steps (solving for the parameter value of each point) are usually more expensive than those for the convex segment method (following pointers and locating points).

7. Complexity. In this section, we analyze the complexity of the convex segment method of sorting. We base our complexity analysis on the RAM model, where basic arithmetic operations are of unit cost [3].

It must be emphasized that the n of the following analysis is the degree of the curve. This makes the analysis fundamentally different from those that we are familiar with, such as $O(n \log n)$ for sorting numbers (where n is the number of points) or $O(n \log \log n)$ for triangulating a simple polygon (where n is the number of edges of the polygon). For example, in the following analysis, n is the constant 1 for all polygons. As a result, the complexity of an operation such as the convex decomposition of an algebraic curve can be misleading, since it is very easy (although wrong) to compare it with familiar complexities of discrete (rather than continuous) algorithms such as number sorting or polygon manipulation.

It should also be noted that the following analysis is pessimistic. The worst-case time will be reached only by the most pathological curves: the time to decompose and sort points on curves that arise in practice in geometric modeling is much more reasonable. This is a major reason why complexity analyses of solid modeling algorithms are rare and more valuable for their insight into the algorithm than their reflection of its performance. For example, a typical endpoint will lie on the boundary of a single-segment cell and its partner will be computed in $O(1)$, not $O(k\beta[n])$, time (see below). In particular, the $O(n^6\beta[n])$ term in Theorem 7.1 will often be closer to n^3 than $n^6\beta[n]$ in reality. This observation has been borne out in practice, with the testing of the algorithms on various curves (see § 8). The efficiency will be even further improved by the fact that the singularities and flexes of the curve, which are important to other geometric algorithms, may already be available in many cases.

7.1. Complexity of convex decomposition.

THEOREM 7.1. *A curve of degree n (a curve whose defining polynomial is degree n) can be decomposed into convex segments in $O(\beta[n^2] + n^2\beta[\mathrm{MAX} * n] + n^6\beta[n])$ time, where $\beta[n]$ is the time required to find the real roots of a univariate polynomial equation of degree n, and* MAX *is the maximum number of quadratic transformations*

that are necessary to decompose any singularity of the curve into simple points.[14]

Proof. Computation of singularities, flexes. Consider the curve $f(x,y) = 0$ of degree n. Its singularities are found by solving the simultaneous system of equations $\{f_x = 0, f_y = 0, f = 0\}$. One method is to use resultants [28]. The resultant of two polynomials with respect to the variable x_n is a polynomial whose roots are the projection onto the hyperplane $x_n = 0$ of the intersections of the two polynomials. Let X (respectively, Y) be the real roots of the resultant of f_x and f_y with respect to y (respectively, x), which is a univariate polynomial in x (respectively, y) of degree $O(n^2)$.[15] X (respectively, Y) is the collection of abscissae (respectively, ordinates) of the solution set of $\{f_x = 0, f_y = 0\}$. X (and Y) can be computed in $O(n^4 \log^3 n + \beta[n^2])$ time, since the resultant of a pair of polynomials of degree at most n in r variables can be computed in $O(n^{2r} \log^3 n)$ time [9]. The singularities of the curve are $\{ (x,y) : x \in X,\ y \in Y$ and $f(x,y) = f_x(x,y) = f_y(x,y) = 0 \}$. This pairwise substitution takes $O(n^6)$ time, since X and Y are each of size $O(n^2)$ and the evaluation of an equation of degree n requires $O(n^2)$ time. Hence, all singularities of the curve can be computed in $O(\beta[n^2] + n^6)$ time. With similar techniques, the flexes can also be computed in $O(\beta[n^2] + n^6)$ time.

Computation of their tangents. Recall from § 5.5 that the tangents at a singularity (a,b) are computed by translating the singularity to the origin and factoring the polynomial consisting of the terms of lowest degree of the translated $f(x,y)$ into linear factors. (For example, the lines $y - x = 0$ and $y + x = 0$ are the tangents of the curve $y^2 + x^3 - x^2 = 0$ at the origin.) A translation is simply a linear substitution $x_t = x - a$, $y_t = y - b$, which takes $O(n^4)$ time for a bivariate equation of degree n. The factorization of a homogeneous bivariate polynomial is equivalent to the solution of a univariate polynomial. Therefore, the computation of the tangents at a singularity requires $O(n^4 + \beta[n])$ time. A curve of degree n has at most $O(n^2)$ singularities [28], so all of the tangents at singularities can be computed in $O(n^6 + n^2\beta[n])$ time. The computation of the tangent at a flex is easier, only involving the $O(n^2)$ operation of bivariate (or homogeneous trivariate) polynomial evaluation (§ 5.5). A curve of degree n also has at most $O(n^2)$ flexes [28], so all of the tangents at flexes can be computed in $O(n^4)$ time.

Computation of intersections of singularity/flex tangents with curve. The intersections of the singularity/flex tangents with the curve are needed to create the convex decomposition. Consider the number of tangents. There are at most $O(n^2)$ tangents at flexes. A curve of order n has at most $(n-1)(n-2)/2$ double points, where a singularity of multiplicity t counts as $t(t-1)/2$ double points and has $O(t)$ tangents [28]. Consequently, there are $t/t(t-1)/2 < 2$ tangents per double point, or at most $O(n^2)$ singularity tangents. The intersection of a tangent with the curve involves a linear substitution and a solution of the resulting polynomial, thus $O(n^4 + \beta[n])$ time or $O(n^6 + n^2\beta[n])$ for all tangents. Note that the $O(n^2)$ tangents generate $O(n^3)$ endpoints on the curve, since each tangent intersects the curve in at most n points (Bezout's theorem).

[14] MAX is 1 if each singularity has distinct tangents, and MAX will usually be 1 or 2 in geometric modeling applications.

[15] Since singularities at infinity are not of interest, those roots in X (respectively, Y) that cause the terms of highest degree of $\{f_x = 0, f_y = 0\}$ to simultaneously vanish are not of interest. (The terms of highest degree of a polynomial are intimately related to its solutions at infinity, since they dominate the polynomial as solutions get large.) Therefore, before computing the roots of the resultant, the greatest common denominator of the leading term polynomials of f_x and f_y is computed and divided out of the resultant, all in $O(n \log^2 n)$ time [3].

Refinement of singularities and infinite segments. A singularity of multiplicity t is refined into $O(2t)$ endpoints, meaning $2t/(t(t-1)/2) \leq 4$ refined endpoints per double point, or a total of $O(n^2)$ refined endpoints at singularities. Thus, the number of endpoints of convex segments (and thus the number of convex segments) remains $O(n^3)$ after refinement. Consider the time that is required to refine the singularities. Each singularity is translated to the origin and subjected to quadratic transformations (perhaps translating the singularity back to the origin after certain quadratic transformations). $O(n^2)$ quadratic transformations are sufficient to reduce all of the singularities to simple points, since the singularities of a curve of degree n account in total for $O(n^2)$ double points and the application of each quadratic transformation removes at least one double point, in a global amortized counting [2]. We have seen that the translation of a curve requires $O(n^4)$ time, amounting to a total of $O(n^6)$ translation time. Each quadratic substitution $x = x_1$, $y = x_1 y_1$ takes $O(n^2)$ time (there are $O(n^2)$ terms in the original equation of the curve). Therefore, all of the quadratic transformations take $O(n^4)$ time.

During the reduction of a singularity to simple points, each quadratic transformation can increase the degree of the curve's equation, since $x^i y^j$ becomes $x^i(x^{j-d}y^j) = x^{i+j-d}y^j$, where d is the multiplicity of the singularity.[16] In other words, the degree of the polynomial can increase by $O(j)$, where j is the highest degree of y in any term of the polynomial undergoing quadratic transformation. Since $j = n$ for the polynomial of the original curve and the y-degree of every term remains invariant under quadratic transformation (and does not increase under translation of the curve either), the degree of the polynomial can only increase by $O(n)$ with each quadratic transformation. Therefore, by the end of the reduction of a singularity to simple points, the curve's equation can be at most degree $O(\mathrm{MAX} * n)$.

Finally, after a quadratic transformation where the multiplicity of the singularity drops, we compute the intersections of the new curve of degree i with the y-axis, which takes $\beta[i]$ time. Again, since this is computed after at most $O(n^2)$ quadratic transformations, the total time taken by all of the intersection computations is at most $O(n^2\beta[\mathrm{MAX} * n])$ time. We conclude that a bound on the time for refining the convex segment endpoints at singularities is $O(n^6 + n^2\beta[\mathrm{MAX} * n])$. The refinement of infinite segments is simple compared to the refinement of singularities.

Pairing endpoints. Consider the time required to compute the partners of the $O(n^3)$ endpoints. The dominating expense is the computation of the set $R(W_1)$ of Theorem 5.5 for each endpoint W_1. It takes $O(k\beta[n])$ time to compute $R(W_1)$ for an endpoint in a cell with k endpoints, $O(k^2\beta[n])$ time to compute $R(W_1)$ for every endpoint in a cell with k endpoints, and $O(\sum k_i^2\beta[n])$ time to compute $R(W_1)$ for every endpoint in every cell, where k_i is the number of endpoints in cell C_i and the sum is over all cells C_i. Since $\sum k_i = O(n^3)$, $O(\sum k_i^2\beta[n]) = O(n^6\beta[n])$. Therefore, partner computation takes $O(n^6\beta[n])$ time. □

7.2. Complexity of sorting. We now consider the complexity of sorting points along a curve after its convex decomposition is available. This sorting is usually very efficient, because the traversal of a curve by convex segments has been reduced to the traversal of a doubly linked list, and it is usually simple to find the points on each convex segment. Once again, the following worst-case analysis is unrealistically

[16] It might appear that $x^i y^j$ should become $x^i(x^j y^j)$. However, redundant factors must be removed from the polynomial. For example, $x^2 - y^3 = 0$ becomes $1 - xy^3 = 0$, not $x^2 - x^3y^3 = 0$. The equation of a curve with a singularity of multiplicity d at the origin has no terms of degree less than d, so a factor of x^d can always be removed.

pessimistic for geometric modeling applications.

THEOREM 7.2. *After the curve has been decomposed into convex segments, m points on a plane algebraic curve of degree n can be sorted by the convex segment method in $O(mn^3\beta[n] + m\log m)$ time.*

Proof. The dominating expense of sorting is to locate every point on a convex segment, since the convex segments are already implicitly sorted (by endpoint pairing) and the sorting of points along a convex segment is simple (by Theorem 4.1, it is equivalent to the $O(k\log k)$ operation of finding and sorting a set of angles). A point can easily be located in the proper cell of the cell partition, as follows. A vector of size $O(n^2)$ is associated with each of the m points and each cell: this vector specifies the side (inside or outside) of each singularity/flex tangent that the point or cell lies on. A point lies in a cell if and only if their two vectors match.[17] Therefore, the only potentially challenging step is locating the convex segment that contains the point.

In the worst case, it requires $O(k\beta[n])$ time to compute the set $S(x)$ of Theorem 6.1 for a point in a cell with k endpoints, since the intersection of line segments with the curve is required. There are $O(n^3)$ endpoints, so point location requires $O(n^3\beta[n])$ time per point and $O(mn^3\beta[n])$ time for all points.[18] After adding $O(m\log m)$ time for sorting the points along the convex segments, the convex segment method requires worst-case $O(mn^3\beta[n]+m\log m)$ time to sort m points by traversing $O(n^3)$ convex segments. □

8. Execution times. This section presents execution times for the sorting of some representative curves by the convex segment and parameterization methods. These empirical results are a good complement to the complexity analysis of § 7, since they capture the expected case, rather than the worst-case, behavior of the methods. The source code was written in Common Lisp and execution times are in seconds on a Symbolics Lisp Machine, not including time for disk faults and garbage collection. Times for the convex segment method are the average of 12 trials, while times for the parameterization method are the average of three trials. Preprocessing time is the time required to create the cell partition and find the partners of all of the endpoints. Five curves are examined: two rational cubics and three nonrational quartics.

We do not consider the time required to find a parameterization of the curve or to find the flexes and singularities of the curve. Each of these computations is a preprocessing step that is entirely independent of sorting, and often the parameterization, singularities, and flexes of a curve will already be available. Moreover, the computation of a curve's parameterization is of approximately the same complexity as the computation of a curve's singularities and flexes, so our comparison of sorting methods should not be biased.

The first example illustrates the superiority of the convex segment method: even when the preprocessing time is added to the sorting time, it is more efficient. Also notice that the rate of growth of the convex segment method is much smaller. The

[17] The vector of a cell need not, and will not, be complete. Only the entries for the cell's walls are necessary.

[18] Observe the worst-case pessimism of this analysis, which assumes that each point is located in a cell with $O(n^3)$ endpoints. It is unlikely that there are $O(n^3)$ real endpoints, since many of the n intersections of a singularity/flex tangent with the curve will be complex. Moreover, it is extremely unlikely that $O(n^3)$ of these endpoints lie in the same cell and that all points to be sorted lie in such a cell. Finally, it is very unlikely that each point location in this cell will require the more expensive conditions of $S(x)$ to test the membership of all $O(n^3)$ endpoints in $S(x)$.

inferiority of the tracing method (see the end of § 3) is obvious from this example, and we do not consider it further.

Example 8.1. A semi-cubical parabola.
Equation of the curve: $27y^2 - 2x^3 = 0$.
Preprocessing time: 0.27 seconds.
Parameterization: $\{x(t) = 6t^2,\ y(t) = 4t^3\ :\ t \in (-\infty, +\infty)\}$.

TABLE 1
Semi-cubical parabola.

Number of sortpoints	1	2	6
Convex segment	.01	.03	.03
Convex segment + preprocessing	.28	.30	.30
Parameterization	.47	.63	1.04
Tracing	3.14	2.89	4.77

The second example illustrates the tradeoff between a very fast sort that requires preprocessing (convex segment method) and a moderately fast sort that does not require preprocessing (parameterization method).

Example 8.2. Folium of Descartes.
Equation of the curve: $x^3 + y^3 - 15xy = 0$.
Preprocessing time: 2.81 seconds.
Parameterization: $\{x(t) = \frac{15t}{1+t^3},\ y(t) = \frac{15t^2}{1+t^3}\ :\ t \in (-\infty, +\infty)\}$.

TABLE 2
Folium of Descartes.

Number of sortpoints	1	2	5	9
Convex segment	0.01	0.01	0.05	0.04
Convex segment + preprocessing	2.82	2.82	2.85	2.85
Parameterization	1.01	1.07	1.76	3.17

The remaining three curves are nonrational, so they are only sorted with the convex segment method.

Example 8.3. Devil's curve (with several connected components).
Equation of the curve: $y^4 - 4y^2 - x^4 + 9x^2 = 0$.
Preprocessing time: 2.20 seconds.

TABLE 3
Devil's curve.

Number of sortpoints	1	4	7
Convex segment	0.09	0.09	0.10
Convex segment + preprocessing	2.29	2.29	2.30

Example 8.4. Limacon.
Equation of the curve: $x^4 + y^4 + 2x^2y^2 - 12x^3 - 12xy^2 + 27x^2 - 9y^2 = 0$.
Preprocessing time: 4.62 seconds.

TABLE 4
Limacon.

Number of sortpoints	2	5	8
Convex segment	.09	.30	.55
Convex segment + preprocessing	4.70	4.92	5.17

Example 8.5. Cassinian oval.
Equation of the curve: $x^4 + y^4 + 2x^2y^2 + 50y^2 - 50x^2 - 671 = 0$.
Preprocessing time: 5.36 seconds.

TABLE 5
Cassinian oval.

Number of sortpoints	2	4	6
Convex segment	.14	.17	.19
Convex segment + preprocessing	5.50	5.53	5.55

We finish this section by considering the relative merits of the parameterization and convex segment methods of sorting. Certain curves cannot, or should not, be sorted by the parameterization method: curves that do not possess a rational parameterization and curves for which a rational parameterization cannot be efficiently obtained. Therefore, the convex segment method is often the only viable way to sort points along an algebraic curve.

For those curves that can be sorted in either way, the convex segment method is generally far more efficient than the parameterization method at the actual sorting of the points. However, the parameterization method does not have the expense of preprocessing that the convex segment method does. Therefore, when only a few points need to be sorted (over the entire lifetime of the curve) and the sorting of these points must be done soon after the definition of the (rational) curve, the parameterization method will usually be the method of choice. (However, we have seen an example where the convex segment method is superior to parameterization even when we include preprocessing time.) The expense of preprocessing will be warranted whenever sorting time is a valuable resource, as in a real-time application, or when the number of points that will be sorted is large. The convex segment method will also be preferable when the curve is defined long before it is ever sorted (as with a complex solid model that requires several days, weeks, or even months to develop), since the preprocessing can be done at any time that processing time becomes available before the sort. We conclude that the convex segment method is an effective new method for sorting points along an algebraic curve, and that in many situations it is either the only or the best method.

9. Conclusions. We have developed a new method of sorting points along an algebraic curve that is superior to the conventional methods of sorting. Many curves that could not be sorted, or that could only be sorted slowly, can now be sorted efficiently. The development of our new method has also illustrated how an algebraic curve can be decomposed into convex segments, how to locate points on segments of algebraic curves, and how to decide whether two points lie on the same connected component of an algebraic curve. These results are of interest in a more general

context than sorting.

This work is one of the first solutions of a computational geometry problem that is applicable to curves of arbitrary degree. Methods are usually restricted to curves/surfaces of some specific or bounded degree, such as polygons/polyhedra or quadrics. The creation and manipulation of curves and surfaces is of major importance to geometric modeling. A sophisticated geometric modeling system should offer a rich collection of tools to aid this manipulation. This paper has been an examination of one of these tools. The progress of geometric modeling depends upon the development of more tools and upon the extension of more computational geometry algorithms from polygons to curves and surfaces of higher degree.

10. Appendix.

10.1. Constructing artificial walls. Section 5.2 defined the artificial walls of an unbounded cell, and discussed the properties that these walls must satisfy. We now give an algorithm for constructing the artificial walls of an unbounded cell C. Recall that artificial walls must be chosen so that (i) they intersect any infinite convex segment in C exactly once, (ii) they do not intersect any finite convex segments, and (iii) the resulting artificially bounded cell is convex. Let P and Q be two extremal points on the boundary of C, such that all original endpoints on C's cell boundary lie between P and Q (Fig. 28). The artificial boundary that we construct shall consist of two or three walls: a wall touching P, a wall touching Q, and occasionally a wall joining these two walls. We call these added walls W_P, W_Q, and W_{PQ}, respectively. We must show how to create W_P, W_Q, and W_{PQ}.

Consider the set of tangents from points of the curve inside C. Let T be the finite subset consisting of tangents that intersect P and make a larger angle than $\vec{PQ}$ with P's wall (measuring from the side of P's wall that contains endpoints). (A method for finding these tangents is discussed below.) If $T = \emptyset$ and P and Q lie on different walls, W_P can be $\overline{PQ}$ (Fig. 28(b)), because if a curve segment crosses $\overline{PQ}$ twice, then the tangent of some point outside of $\overline{PQ}$ intersects P and Q. If $T = \emptyset$ and P and Q lie on the same wall, then W_P can be the normal to P's wall. If $T \neq \emptyset$, W_P will be the tangent in T that makes the largest angle with P's wall, but rotated about P so that it makes an even larger angle with P's wall, thus avoiding the creation of a redundant new endpoint at the point of tangency with the curve (Fig. 28(a)). W_Q is created in an entirely analogous way. The only wall remaining to define is W_{PQ}, the wall joining W_P and W_Q. If W_P and W_Q intersect inside the cell C, then no W_{PQ} is needed. Otherwise, let V be the tangents (from points of the curve inside C) that are parallel to $\overline{PQ}$. If $V = \emptyset$, let W_{PQ} be an arbitrary line segment connecting W_P and W_Q. If $V \neq \emptyset$, W_{PQ} will be a segment of the tangent in V that lies furthest from $\overline{PQ}$, translated some distance away from $\overline{PQ}$ (Fig. 28(c)).

As part of the above construction, it is necessary to compute the tangents of nonsingular points that intersect a given point, as well as those that are parallel to a given line. The tangent at the nonsingular point α of the plane curve $f(x, y, w) = 0$ (where w is the homogeneous coordinate, placing the curve in projective space) is $\sum_{i=1}^{3} f_{x_i}(\alpha)x_i$, where $x_1 = x$, $x_2 = y$, $x_3 = w$, and f_{x_i} is the derivative of f with respect to x_i [28]. Thus, the nonsingular points $\alpha = (\alpha_1, \alpha_2, 1)$ of a curve $f(x, y, w) = 0$ whose tangents intersect an arbitrary point $P = (p_1, p_2, p_3) = (p_1, p_2, 1)$ of the plane can be computed by solving the pair of equations $\{f(\alpha) = 0, \sum_{i=1}^{3} f_{x_i}(\alpha)p_i = 0\}$ for α_1 and α_2, and eliminating singularities. For the second problem, note that the slope of the tangent at α is $-f_x(\alpha)/f_y(\alpha)$, unless the line is vertical in which case $f_y(\alpha) = 0$. Thus, the nonsingular points $\alpha = (\alpha_1, \alpha_2, 1)$ of a curve $f(x, y, w) = 0$ whose tangents

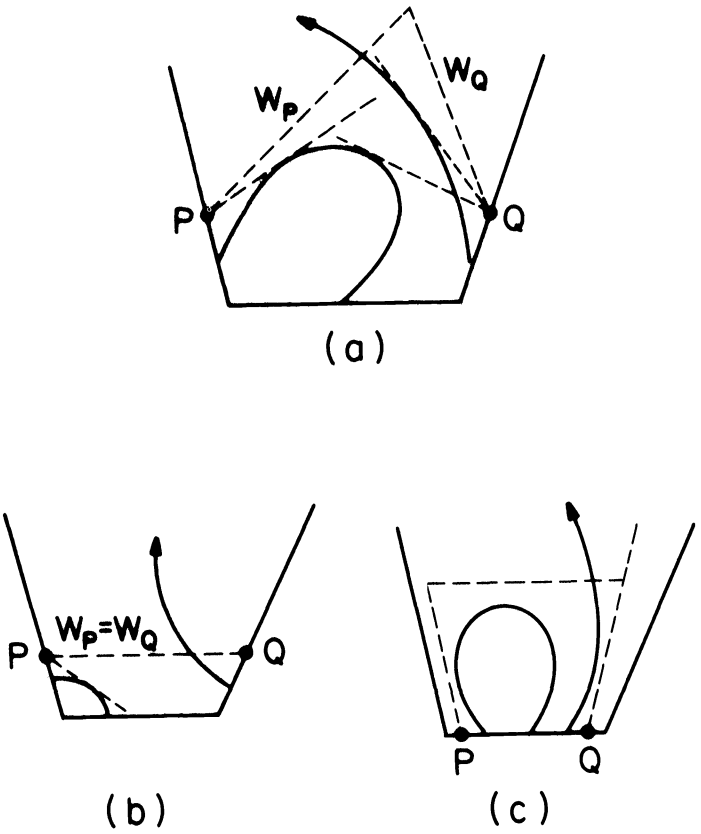


FIG. 28. *Constructing artificial walls.*

are parallel to the vector (a, b) can be computed by solving the pair of equations $\{f(\alpha) = 0,\ af_x(\alpha) + bf_y(\alpha) = 0\}$ for α_1 and α_2 and eliminating singularities.

10.2. Three key lemmas. We present three important lemmas in this section. The first lemma is used in the proof of the point location theorem (Theorem 6.1), as well as the proof of the second lemma below. The second lemma is crucial in the proofs of both of the main theorems (Theorems 5.5 and 6.1). The third lemma is used in Theorem 5.5.

LEMMA 10.1. *Let x be any nonsingular point of the curve F in cell C, and let* $\mathcal{S}(x) = \{$*endpoints W in C |*

(1) $\#\{P \in \overline{xW} \cap F : P \text{ faces } x\} = \#\{P \in \overline{xW} \cap F : P \text{ faces } W\}$,

(2) *For all* $\alpha \in \overline{xW}$, $\#\{P \in \overline{x\alpha} \cap F : P \text{ faces } x\} \leq \#\{P \in \overline{x\alpha} \cap F : P \text{ faces } W\}\}$.

(*Note that $\mathcal{S}(x)$ is a superset of the $S(x)$ of Theorems* 5.5 *and* 6.1.) *If $s \in \mathcal{S}(x)$, then neither s's convex segment nor x's convex segment can cross $\overline{sx}$.*

Proof. Suppose that s's convex segment crosses $\overline{sx}$ at y (Fig. 29). Then, using Lemma 5.3, $\#\{P \in \overline{sy} \cap F : P \text{ faces } s\} = \#\{P \in \overline{sy} \cap F : P \text{ faces } y\} = \#\{P \in \overline{sy} \cap F : P \text{ faces } x\}$. Since y faces s and $y \notin \overline{sy}$, if we choose $\alpha \in \overline{yx}$ such that $\overline{y\alpha}$ does not contain any intersections with the curve, $\#\{P \in \overline{s\alpha} \cap F : P \text{ faces } s\} > \#\{P \in \overline{s\alpha} \cap F : P \text{ faces } x\}$. Since $s \in \mathcal{S}(x)$, we know that $\#\{P \in \overline{sx} \cap F : P \text{ faces } s\} = \#\{P \in \overline{sx} \cap F : P \text{ faces } x\}$. Therefore, the above inequality becomes $\#\{P \in \overline{\alpha x} \cap F : P \text{ faces } s\} < \#\{P \in \overline{\alpha x} \cap F : P \text{ faces } x\}$, which contradicts $s \in \mathcal{S}(x)$ (condition (2)).

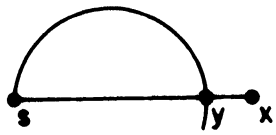


FIG. 29. *s's segment cannot cross $\overline{sx}$.*

The proof for x's convex segment is similar. Suppose that x's segment crosses $\overline{sx}$ at y. Then $\#\{P \in \overline{xy} \cap F : P \text{ faces } x\} = \#\{P \in \overline{xy} \cap F : P \text{ faces } y\} = \#\{P \in \overline{xy} \cap F : P \text{ faces } s\}$. Since y faces x, if we choose $\alpha \in \overline{ys}$ such that $\overline{y\alpha}$ does not contain any intersections with the curve, $\#\{P \in \overline{x\alpha} \cap F : P \text{ faces } x\} > \#\{P \in \overline{x\alpha} \cap F : P \text{ faces } s\}$, in contradiction of $s \in \mathcal{S}(x)$. □

LEMMA 10.2. *Let x be any nonsingular point of the curve F in cell C, and let z be any point such that z lies on x's convex segment and $\widehat{xz}$ does not contain any endpoints in its interior (Fig.* 30). *Let* $S(x) = \{$*endpoints W in C |*

(1) *W faces x,*

(2) $\#\{P \in \overline{xW} \cap F : P \text{ faces } x\} = \#\{P \in \overline{xW} \cap F : P \text{ faces } W\}$,

(3) *For all* $\alpha \in \overline{xW}$, $\#\{P \in \overline{x\alpha} \cap F : P \text{ faces } x\} \leq \#\{P \in \overline{x\alpha} \cap F : P \text{ faces } W\}\}$.

(*Note that $S(x)$ is a superset of the $S(x)$ of Theorems* 5.5 *and* 6.1.) *Then $\vec{xs}$ does not cross $\widehat{xz} \setminus \{z\}$ for any $s \in S(x)$.*

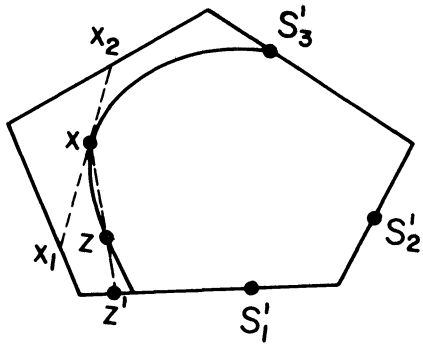


FIG. 30. $\vec{xs}$ *does not cross* $\widehat{xz}$.

Proof. Let $s \in S(x)$. Suppose, for the sake of contradiction, that $\vec{xs}$ crosses $\widehat{xz} \setminus \{z\}$ at y. There are three cases to consider: $y \in \overline{xs}$, $y = s$, and $y \notin \overline{xs}$ (i.e., $s \in \overline{xy}$). $y \in \overline{xs}$ contradicts Lemma 10.1. $y = s$ is also contradictory, since $\widehat{xz}$ does not contain any endpoints in its interior. $s \in \overline{xy}$ is the only nontrivial case (Fig. 31). Recall an argument used at the beginning of Lemma 5.3: since $\widehat{xy}$ is part of a convex segment lying in a cell and x and y are nonsingular points, the points of entry and departure of the curve into the closed region bounded by $\overline{xy}$ and $\widehat{xy}$ must be along $\overline{xy}$ and must pair up into couples. In particular, s must pair with another point, say $t \in \overline{xy}$. We shall use t to develop a contradiction of Lemma 5.3 for the convex segment $\widehat{xy}$. Since $\widehat{st}$ is convex, s faces t; and since $s \in S(x)$, s also faces x. Therefore, $t \in \overline{xs}$. Since $s \in S(x)$,

$$\#\{P \in \overline{xs} \cap F : P \text{ faces } x\} = \#\{P \in \overline{xs} \cap F : P \text{ faces } s\}.$$

Noting that $\overline{xs} = \overline{xt} \cup \overline{ts} \cup \{t\}$ and t faces s, this becomes

$$\begin{aligned} &\#\{P \in \overline{xt} \cap F : P \text{ faces } x\} + \#\{P \in \overline{ts} \cap F : P \text{ faces } x\} + 0 \\ = \; &\#\{P \in \overline{xt} \cap F : P \text{ faces } s\} + \#\{P \in \overline{ts} \cap F : P \text{ faces } s\} + 1. \end{aligned}$$

Moreover, by Lemma 5.3,

$$\begin{aligned} \#\{P \in \overline{ts} \cap F : P \text{ faces } s\} &= \#\{P \in \overline{ts} \cap F : P \text{ faces } t\} \\ &= \#\{P \in \overline{ts} \cap F : P \text{ faces } x\}. \end{aligned}$$

Upon cancelling terms in the above equation, we conclude that

$$\begin{aligned} \#\{P \in \overline{xt} \cap F : P \text{ faces } x\} &> \#\{P \in \overline{xt} \cap F : P \text{ faces } s\} \\ &= \#\{P \in \overline{xt} \cap F : P \text{ faces } y\}. \end{aligned}$$

But this contradicts condition (3) of Lemma 5.3 ($X = x$, $Y = y$). These contradictions lead us to conclude that $\vec{xs}$ does not cross $\widehat{xz} \setminus \{z\}$. □

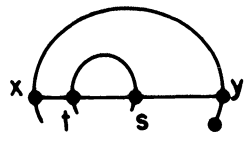


FIG. 31. $s \in \overline{xy}$.

COROLLARY 10.3. *Let x, z, C and $S(x)$ be as in the previous lemma. Let z' be the intersection of $\vec{xz}$ with the boundary of C, let $S'(x) = \{s' : s'$ is the intersection of $\vec{xs}$ with the boundary of C, $s \in S(x)\}$, and let x_1, x_2 be the intersections of x's tangent with the cell boundary (Fig. 30). If $S'(x)$ is sorted along the boundary of C from x_1 to x_2, then z' is either the first or the last element.*

LEMMA 10.4 [18, p. 119]. *Let W_1 and W_2 be partners. If W_2 lies on W_1's tangent, then W_1 must be a flex. In other words, if W_2 lies on W_1's tangent, then W_1 and W_2 lie on the same cell wall.*

10.3. Computing the infinite direction from a point. In order to compute the type of an infinite convex segment (§ 6.1), it is necessary to compute the infinite direction from a point of that infinite segment. As seen in Theorem 6.1, for our purposes the point will either be an artificial endpoint or a point outside the bounded cell. The following lemma shows how to perform this computation.

LEMMA 10.5. *Let x be a point that lies on an infinite convex segment, and let T_x^+ and T_x^- be the two rays from x along its tangent. Let C be the unbounded cell (before refinement) that contains x, and let $C' \subset C$ be the associated artificially bounded cell (after refinement).*

(i) *If x is an artificial endpoint, then T_x^+ is the infinite direction from x if T_x^+ points into C'.*

(ii) *If x lies outside C', then T_x^+ is the infinite direction from x if and only if*
   (a) *T_x^+ does not intersect the boundary of C, or*
   (b) *Both T_x^+ and T_x^- intersect the boundary of C, but T_x^+ does not enter C'.*

Proof. Part (i) is clear, so assume that x lies outside C'. It is simple to show that at least one of the rays must intersect the boundary of C, and we leave this as an exercise for the reader.

(a) Suppose that T_x^+ does not intersect the boundary of C and T_x^- does. If T_x^+ is the finite direction, then the convex segment must travel to a cell wall after leaving x along T_x^+ (Fig. 32). But, in so doing, it will block the other half of the convex segment (leaving x along T_x^-) from traveling to infinity. Therefore, T_x^+ must be the infinite direction.

(b) Suppose that both T_x^+ and T_x^- intersect the boundary of C. Let T_x be the entire tangent at x, which divides C into a bounded and an unbounded half. Let

$\widehat{P\infty}$ be x's infinite convex segment before refinement. Since $\widehat{P\infty}$ must lie entirely on one side of x's tangent (by convexity) and in an unbounded region (because $\widehat{P\infty}$ is infinite), it must lie on the unbounded half. In particular, the unbounded half must contain the original endpoint P, which lies in C' on an original wall. Thus, T_x must enter C'. If both T_x^+ and T_x^- enter C', then all of T_x (and in particular x) must lie in C', by the convexity of C'. Thus, since $x \notin C'$, exactly one of T_x^+ and T_x^- enters C'. The unique ray that enters C' must be the finite direction; otherwise the segment leaving x in the infinite direction will be blocked, just as in (a). □

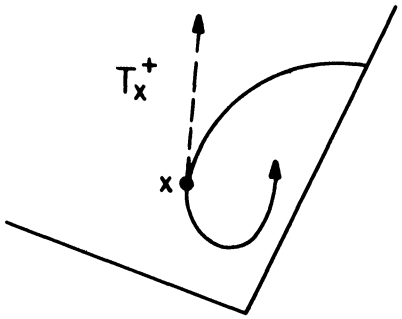


FIG. 32. T_x^+ *cannot be the finite direction.*

Acknowledgments. This work formed part of the Ph.D. thesis of John Johnstone, who is very grateful for the guidance of his advisor, John Hopcroft. The thorough and insightful reading of the main referee is also much appreciated.

REFERENCES

[1] S. ABHYANKAR, *Desingularization of Plane Curves*, American Mathematical Society, Providence, RI, 1983, pp. 1–45.

[2] S. ABHYANKAR AND C. BAJAJ, *Automatic parameterization of rational curves and surfaces 3: Algebraic plane curves*, Comput. Aided Geom. Des., 5 (1988), pp. 309–321.

[3] A. AHO, J. HOPCROFT, AND J. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[4] D. S. ARNON, *Topologically reliable display of algebraic curves*, J. Comput. Graphics, 17 (1983), pp. 219–227.

[5] T. ASANO, T. ASANO, AND H. IMAI, *Partitioning a polygonal region into trapezoids*, Tech. Report Res. Mem. RMI84-03, Department of Mathematics, Engineering, and Instrumentation Physics, University of Tokyo, Tokyo, Japan, 1984.

[6] D. AVIS AND G. T. TOUISSANT, *An efficient algorithm for decomposing a polygon into star-shaped components*, Pattern Recognition, 13 (1981), pp. 395–398.

[7] C. BAJAJ, *Geometric modeling with algebraic surfaces*, in Mathematics of Surfaces III, Oxford University Press (to appear).

[8] C. BAJAJ, C. HOFFMANN, J. HOPCROFT, AND B. LYNCH, *Tracing surface intersections*, Comput. Aided Geom. Des., 5 (1988), pp. 285–307.

[9] C. BAJAJ AND A. ROYAPPA, *GANITH: An algebraic geometry package*, Tech. Report CSD-TR-914, Department of Computer Science, Purdue University, West Lafayette, IN.

[10] J. F. CANNY, *The Complexity of Robot Motion Planning*, MIT Press, Cambridge, MA, 1987.
[11] B. CHAZELLE AND D. P. DOBKIN, *Optimal convex decompositions*, in Computational Geometry, G. T. Toussaint, ed., North-Holland, New York, 1985, pp. 63–133.
[12] B. CHAZELLE AND J. INCERPI, *Triangulation and shape-complexity*, ACM Trans. Graphics, 3 (1984), pp. 135–152.
[13] G. E. COLLINS, *Quantifier elimination for real closed fields by cylindrical algebraic decomposition*, in Proc. 2nd GI Conference on Automata Theory and Formal Languages, Lecture Notes in Computer Science Vol. 35, Springer-Verlag, Berlin, 1975, pp. 134–183.
[14] M. DO CARMO, *Differential Geometry of Curves and Surfaces*, Prentice–Hall, Englewood Cliffs, NJ, 1976.
[15] M. R. GAREY, D. S. JOHNSON, F. P. PREPARATA, AND R. E. TARJAN, *Triangulating a simple polygon*, Inform. Process. Lett., 7 (1978), pp. 175–179.
[16] S. HERTEL AND K. MEHLHORN, *Fast triangulation of simple polygons*, in Proc. FCT'83, Borgholm, Lecture Notes in Computer Science, Springer-Verlag, Berlin, 1983, pp. 207–218.
[17] C. HOFFMANN AND J. HOPCROFT, *The potential method for blending surfaces and corners*, in Geometric Modeling: Algorithms and New Trends, G. Farin, ed., Society for Industrial and Applied Mathematics, Philadelphia, PA, 1987, pp. 347–365.
[18] J. K. JOHNSTONE, *The sorting of points along an algebraic curve*, PhD thesis, Department of Computer Science, Cornell University, Ithaca, NY, 1987.
[19] J. M. KEIL, *Decomposing polygons into simpler components*, PhD thesis, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, 1983.
[20] D. KIRKPATRICK, *Optimal search in planar subdivisions*, SIAM J. Comput., 12 (1983), pp. 28–35.
[21] J. D. LAWRENCE, *A Catalog of Special Plane Curves*, Dover, New York, 1972.
[22] W. M. NEWMAN AND R. F. SPROULL, *Principles of Interactive Computer Graphics*, McGraw-Hill, New York, 1979.
[23] J. O'ROURKE, *The complexity of computing minimum convex covers for polygons*, in Proc. 20th Annual Allerton Conference on Communication, Control and Computing, Monticello, IL, 1982, pp. 75–84.
[24] F. PREPARATA AND M. SHAMOS, *Computational Geometry: An Introduction*, Springer-Verlag, New York, 1985.
[25] J. R. SACK, *An $O(n \log n)$ algorithm for decomposing simple rectilinear polygons into convex quadrilaterals*, in Proc. 20th Annual Allerton Conference on Communication, Control and Computing, Monticello, IL, 1982, pp. 64–74.
[26] R. E. TARJAN AND C. J. V. WYK, *An $O(n \log \log n)$-time algorithm for triangulating a simple polygon*, SIAM J. Comput., 17 (1988), pp. 143–178.
[27] S. B. TOR AND A. E. MIDDLEDITCH, *Convex decomposition of simple polygons*, ACM Trans. Graphics, 3 (1984), pp. 244–265.
[28] R. J. WALKER, *Algebraic Curves*, Springer-Verlag, New York, 1950.

SIAM J. COMPUT.
Vol. 19, No. 5, pp. 968–988, October 1990

013

EFFICIENT PARALLEL ALGORITHMS FOR STRING EDITING AND RELATED PROBLEMS*

ALBERTO APOSTOLICO†, MIKHAIL J. ATALLAH‡, LAWRENCE L. LARMORE§, AND SCOTT MCFADDIN¶

Abstract. The string editing problem for input strings x and y consists of transforming x into y by performing a series of weighted edit operations on x of overall minimum cost. An edit operation on x can be the deletion of a symbol from x, the insertion of a symbol in x or the substitution of a symbol of x with another symbol. This problem has a well-known $O(|x||y|)$ time-sequential solution. Efficient PRAM parallel algorithms for the string editing problem are given. If $m = \min(|x|, |y|)$ and $n = \max(|x|, |y|)$, then the CREW bound is $O(\log m \log n)$ time with $O(mn/\log m)$ processors. The CRCW bound is $O(\log n(\log\log m)^2)$ time with $O(mn/\log\log m)$ processors. In all algorithms, space is $O(mn)$.

Key words. string-to-string correction, edit distances, approximate string searching, spelling correction, longest common subsequence, shortest paths, grid graphs, analysis of algorithms, parallel computation, cascading divide and conquer

AMS(MOS) subject classification. 68Q25

1. Introduction.

One of the major goals of parallel algorithm design for PRAM models is to come up with parallel algorithms that are both *fast* and *efficient*, i.e., that run in polylog time while the product of their time and processor complexities is within a polylog factor of the time complexity of the best sequential algorithm for the problem they solve. This goal has been elusive for many simple problems that are trivially in the class NC (recall that NC is the class of problems that are solvable in $O(\log^{O(1)} n)$ parallel time by a PRAM using a polynomial number of processors). For example, topological sorting of a DAG and finding a breadth-first search tree of a graph are problems that are trivially in NC, and yet it is not known whether either of them can be solved in polylog time with n^2 processors.

This paper gives parallel algorithms for the string editing problem that are both fast and efficient in the above sense. We give a CREW-PRAM algorithm that

* Received by the editors December 10, 1987; accepted for publication (in revised form) January 1, 1990. A preliminary version of these results appeared in the Proceedings of the 26th Annual Allerton Conference on Communication, Control, and Computing, Monticello, Illinois, September 1988, pp. 253–263.

† Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by the Italian and French Ministries of Education, by the Italian National Research Council through IASI-CNR, by National Science Foundation grant CCR-8900305, by National Institutes of Health Library of Medicine grant R01 LM05118, and by the British Research Council grant SERC-E76797.

‡ Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. The research of this author was supported by Office of Naval Research contracts N00014-84-K-0502 and N00014-86-K-0689, by National Science Foundation grant DCR-8451393 with matching funds from AT&T, and by National Institutes of Health Library of Medicine grant R01 LM05118. Part of the research of this author was carried out while he was at the Research Institute for Advanced Computer Science, National Aeronautics and Space Administration (NASA) Ames Research Center, California.

§ Department of Mathematics and Computer Science, University of California, Riverside, California 92521. The research of this author was supported by the University of California, Irvine, California.

¶ Department of Computer Science, Purdue University, West Lafayette, Indiana 47907. The research of this author was carried out while he was at the Research Institute for Advanced Computer Science, National Aeronautics and Space Administration (NASA) Ames Research Center, California.

runs in $O(\log m \log n)$ time with $O(mn/\log m)$ processors, where m (respectively, n) is the length of the shorter (respectively, longer) of the two input strings. We also give a CRCW-PRAM algorithm that runs in $O(\log n(\log\log m)^2)$ time with $O(mn/\log\log m)$ processors. In both algorithms, space is $O(mn)$.

In related work, Ranka and Sahni [17] have designed a hypercube algorithm for $m = n$ that runs in $O(\sqrt{n \log n})$ time with n^2 processors, and have considered time/processor tradeoffs. In independent work, Mathies [15] has obtained a CRCW-PRAM algorithm for the edit distance that runs in $O(\log n \log m)$ time with $O(mn)$ processors if the weight of every edit operation is smaller than a given constant integer. Also independently, Aggarwal and Park have, in [3] and [4], given an $O(\log m \log n)$ time, $O(mn/\log m)$ processor CREW-PRAM algorithm, and an $O((\log\log m)^2 \log n)$ time, $O(mn/(\log\log m)^2)$ processor CRCW-PRAM algorithm. The basic structure of their algorithms is similar to ours, but they use different methods for the "conquer" stage (in particular, they do not use the cascading divide-and-conquer scheme). In the terminology of [3] and [4], the "conquer" stage corresponds to the problem of computing the "tube maxima of a totally monotone $n \times n \times n$ matrix." Within the "conquer" stage, the computation of a single row (as in § 6.1) corresponds in [3] and [4] to the problem of "computing the row maxima of a totally monotone $n \times n$ matrix." We refer the reader to [2]–[4] for the myriad other applications of the "tube maxima" and "row maxima" problems.

Recall that the CREW-PRAM model of parallel computation is the synchronous shared-memory model where concurrent reads are allowed but no two processors can simultaneously attempt to write in the same memory location (even if they are trying to write the same thing). The CRCW-PRAM differs from the CREW-PRAM in that it allows many processors to write simultaneously in the same memory location: in any such common-write contest, only one processor succeeds, but it is not known in advance which one.

The rest of this Introduction reviews the problem, its importance, and how it can be viewed as a shortest-paths problem on a special type of graph.

Let x be a string of $|x|$ symbols on some alphabet I. We consider three *edit operations* on x, namely, *deletion* of a symbol from x, *insertion* of a new symbol in x and *substitution* of one of the symbols of x with another symbol from I. We assume that each edit operation has an associated nonnegative real number representing the *cost* of that operation. More precisely, the cost of deleting from x an occurrence of symbol a is denoted by $D(a)$, the cost of inserting some symbol a between any two consecutive positions of x is denoted by $I(a)$, and the cost of substituting some occurrence of a in x with an occurrence of b is denoted by $S(a, b)$. An *edit script* on x is any consistent (i.e., all edit operations are viable) sequence σ of edit operations on x, and the cost of σ is the sum of all costs of the edit operations in σ.

Now, let x and y be two strings of respective lengths $|x|$ and $|y|$. The *string editing problem* for input strings x and y consists of finding an edit script σ' of minimum cost that transforms x into y. The cost of σ' is the *edit distance from x to y*. In various ways and forms, the string editing problem arises in many applications, notably, in text editing, speech recognition, machine vision and, last but not least, molecular sequence comparison. For this reason, this problem has been studied rather extensively in the past, and forms the object of several papers (e.g., [13], [14], [16], [18]– [20], [23], to list a few). The problem is solved by a serial algorithm in $\Theta(|x||y|)$ time and space, through dynamic programming (cf., for example, [23]). Such a performance represents a lower bound when the queries on symbols of the string are restricted to tests of equality [1],

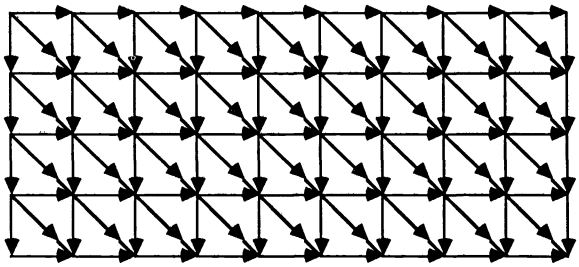


FIG. 1.1. *Example of a* 5×10 *grid DAG.*

[24]. Many important problems are special cases of string editing, including the *longest common subsequence* problem and the problem of *approximate matching* between a pattern string and text string (see [12], [21], and [22] for the notion of approximate pattern matching and its connection to the string editing problem). Needless to say, our solution to the general string editing problem implies similar bounds for all these special cases.

The criterion that subtends the computation of edit distances by dynamic programming is readily stated. For this, let $C(i,j)$, $(0 \leq i \leq |x|,\ 0 \leq j \leq |y|)$ be the minimum cost of transforming the prefix of x of length i into the prefix of y of length j. Let s_k denote the kth symbol of string s. Then $C(0,0) = 0$, and

$$C(i,j) = \min\{C(i-1,j-1) + S(x_i, y_j),\ C(i-1,j) + D(x_i),\ C(i,j-1) + I(y_j)\}$$

for all i,j, $(1 \leq i \leq |x|; 1 \leq j \leq |y|)$. Hence $C(i,j)$ can be evaluated row by row or column by column in $\Theta(|x||y|)$ time [23]. Observe that, of all entries of the C-matrix, only the three entries $C(i-1,j-1)$, $C(i-1,j)$, and $C(i,j-1)$ are involved in the computation of the final value of $C(i,j)$. As was observed in [10], such interdependencies among the entries of the C-matrix induce an $(|x|+1) \times (|y|+1)$ *grid*-directed acyclic graph (grid DAG for short) associated with the string editing problem.

DEFINITION 1. An $l_1 \times l_2$ grid DAG is a directed acyclic graph whose vertices are the $l_1 l_2$ points of an $l_1 \times l_2$ grid, and such that the only edges from grid point (i,j) are to grid points $(i, j+1)$, $(i+1, j)$, and $(i+1, j+1)$.

Figure 1.1 shows an example of a grid DAG and also illustrates our convention of drawing the points such that point (i,j) is at the ith row from the top and jth column from the left. Note that the top-left point is $(0,0)$ and has no edge entering it (i.e., is a *source*), and that the bottom-right point is (m,n) and has no edge leaving it (i.e., is a *sink*).

We now review the correspondence between edit scripts and grid graphs that was observed in [10]. We associate an $(|x|+1) \times (|y|+1)$ grid DAG G with the string editing problem in the natural way: the $(|x|+1)(|y|+1)$ vertices of G are in one-to-one correspondence with the $(|x|+1)(|y|+1)$ entries of the C-matrix, and the *cost* of an edge from vertex (k,l) to vertex (i,j) is equal to $I(y_j)$ if $k = i$ and $l = j-1$; to $D(x_i)$ if $k = i-1$ and $l = j$; and to $S(x_i, y_j)$ if $k = i-1$ and $l = j-1$. We can restrict our attention to edit scripts which are not wasteful in the sense that they do no obviously inefficient moves such as: inserting then deleting the same symbol, or changing a symbol into a new symbol which they then delete, etc. More formally, the only edit scripts considered are those that apply at most one edit operation to a given symbol occurrence. Such edit scripts that transform x into y or vice versa are

in one-to-one correspondence to the weighted paths in G that originate at the source (which corresponds to $C(0,0)$) and end on the sink (which corresponds to $C(|x|,|y|)$). Thus, in order to establish the complexity bounds claimed in this paper, we need only establish them for the problem of finding a shortest (i.e., least-cost) source-to-sink path in an $m \times n$ grid DAG G.

Throughout, the *left boundary* of G is the set of points in its leftmost column. The *right, top,* and *bottom* boundaries are analogously defined. The *boundary* of G is the union of its left, right, top, and bottom boundaries.

The rest of the paper is organized as follows. Section 2 gives a preliminary CREW-PRAM algorithm for computing the length of a shortest source-to-sink path, assuming $m = n$. Section 3 gives an algorithm that uses a factor of $\log m$ fewer processors than the previous one and that will be needed later in our best CREW algorithm (given in §6). Section 4 sketches how to extend the previous algorithm to the case $m \leq n$. Section 5 considers computing the path itself rather than just its length. Section 6 gives our best CREW-PRAM algorithm, which is the main technical result of this paper. Section 7 gives the CRCW-PRAM algorithm. Section 8 concludes the paper.

2. A preliminary algorithm. Throughout this section, $m = n$, i.e., G is an $m \times m$ grid DAG. Let $DIST_G$ be a $(2m) \times (2m)$ matrix containing the lengths of all shortest paths that begin at the top or left boundary of G, and end at the right or bottom boundary of G. In this section we establish that the matrix $DIST_G$ can be computed in $O(\log^3 m)$ time, $O(m^2)$ space, and with $O(m^2/\log m)$ processors by a CREW-PRAM. The preliminary algorithm that achieves this is intended as a "warm-up" for the better algorithms that follow in later sections. The preliminary algorithm works as follows: divide the $m \times m$ grid into four $(m/2) \times (m/2)$ grids A, B, C, D, as shown in Fig. 2.1. In parallel, recursively solve the problem for each of the four grids A, B, C, D, obtaining the four distance matrices $DIST_A$, $DIST_B$, $DIST_C$, $DIST_D$. Then obtain from these four matrices the desired matrix $DIST_G$. The main problem we face, and the main contribution of this paper, is how to perform the "conquer" step efficiently, in parallel.

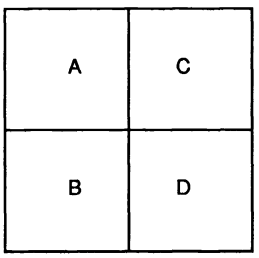


FIG. 2.1. *Illustrating how the problem is partitioned.*

The performance bounds we claimed for this preliminary algorithm would immediately follow if we can show that (i) $DIST_G$ can be obtained from $DIST_A$, $DIST_B$, $DIST_C$, $DIST_D$ in parallel in time $O((q+\log m)\log m)$ and with $O(m^2/q)$ processors, where $q \leq m$ is an integer of our choice, and (ii) the whole problem can be solved sequentially in $O(m^2 \log m)$ time. This is because the time and processor complexities of the overall algorithm would then obey the following recurrences:

$$T(m) \leq T(m/2) + c_1(q + \log m)\log m,$$

$$P(m) \leq \max(4P(m/2), c_2 m^2/q),$$

with boundary conditions $T(\sqrt{q}) = c_3 q \log q$ and $P(\sqrt{q}) = 1$, where c_1, c_2, c_3 are constants. The solutions are $T(m) = O((q + \log m)\log^2 m)$ and $P(m) = O(m^2/q)$. Choosing $q = \log m$ would then establish the desired result.

A sequential $O(m^2 \log m)$ time bound follows from the parallel algorithm we give in §3: it does that much work and hence also translates into a sequential algorithm with this time bound (there is no circularity in the logic: §3 is self-contained). Therefore in the rest of this section, we merely concern ourselves with establishing (i), that is, showing that $DIST_G$ can be obtained from $DIST_A$, $DIST_B$, $DIST_C$, $DIST_D$ in time $O((q + \log m)\log m)$ and with $O(m^2/q)$ processors.

Let $DIST_{A\cup B}$ be the $(3m/2) \times (3m/2)$ matrix containing the lengths of shortest paths that begin on the top or left boundary of $A \cup B$ and end on its right or bottom boundary. Let $DIST_{C\cup D}$ be analogously defined for $C \cup D$. The procedure for obtaining $DIST_G$ performs the following steps:

1) Use $DIST_A$ and $DIST_B$ to obtain $DIST_{A\cup B}$.
2) Use $DIST_C$ and $DIST_D$ to obtain $DIST_{C\cup D}$.
3) Use $DIST_{A\cup B}$ and $DIST_{C\cup D}$ to obtain $DIST_G$.

We only show how step 1) is done, since the procedures for steps 2) and 3) are very similar. First, note that the entries of $DIST_{A\cup B}$ that correspond to shortest paths that begin and end on the boundary of A (respectively, B) are already available in $DIST_A$ (respectively, $DIST_B$), and can therefore be obtained in $O(q)$ time. Therefore we need only worry about the entries of $DIST_{A\cup B}$ that correspond to paths that begin on the top or left boundary of A and end on the right or bottom boundary of B. Assign to every point v on the top or left boundary of A a group of m/q processors. The task of the group of m/q processors assigned to v is to compute the lengths of all shortest paths that begin at v and end on the right or bottom boundary of B. It suffices to show that it can indeed do this in time $O((q + \log m)\log m)$. Observe that:

$$(1) \qquad DIST_{A\cup B}(v,w) = \min\{Dist_A(v,p) + Dist_B(p,w) \mid p \text{ lies on the boundary common to } A \text{ and } B\}.$$

Using (1) to compute $DIST_{A\cup B}(v,w)$ for a given v, w pair is trivial to do in time $O(q + \log(m/q))$ by using $O(m/q)$ processors for each such pair, but that would require an unacceptable $O(m^3/q)$ processor. We have only m/q processors assigned to v for computing $DIST_{A\cup B}(v,w)$ for *all* w on the bottom or right boundary of B. These m/q processors are enough for doing the job in time $O((q + \log(m/q))\log m)$. The procedure is given below.

DEFINITION 2. Let v be any point on the left or top boundary of A, and let w be any point on the bottom or right boundary of B. Let $\theta(v,w)$ denote the *leftmost* p which minimizes the right-hand side of (1). Equivalently, $\theta(v,w)$ is the leftmost point of the common boundary of A and B such that a shortest v-to-w path goes through it.

Define a linear ordering $<_B$ on the m points at the bottom and right boundaries of B, such that they are encountered in increasing order of $<_B$ by a walk that starts at the leftmost point of the lower boundary of B and ends at the top of the right boundary of B. Let L_B be the list of m points on the lower and right boundaries of B, sorted by increasing order according to the $<_B$ relationship. For any $w_1, w_2 \in L_B$, we have the following:

$$(2) \qquad \text{If } w_1 <_B w_2 \text{ then } \theta(v,w_1) \text{ is not to the right of } \theta(v,w_2).$$

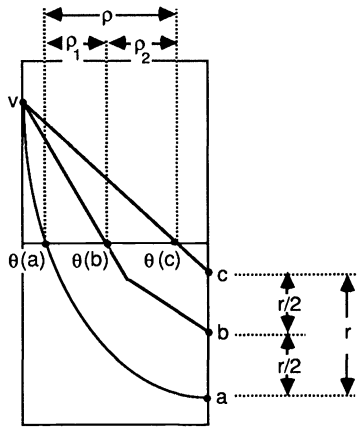


FIG. 2.2. *Illustrating the procedure for computing the function* θ.

A similar property was proved in [9], and in fact Aggarwal and Park [3] have traced this simple observation back to Monge, in 1781. It helps the comprehension of this paper to review the proof of property (2). But before doing so, we sketch how property (2) is used to obtain an $O((q+\log(m/q))\log m)$ time and $O(m/q)$ processor algorithm for computing $DIST_{A\cup B}(v,w)$ for all $w \in L_B$. We henceforth use $\theta(w)$ as a shorthand for $\theta(v,w)$, with v being understood. It suffices to compute $\theta(w)$ for all $w \in L_B$. The procedure for doing this is recursive, and takes as input:

- A particular range of r contiguous values in L_B, say a range that begins at point a and ends at point c, $a <_B c$,
- The points $\theta(a)$ and $\theta(c)$,
- A number of processors equal to $\max\{1,(\rho+r)/q\}$ where ρ is the number of points between $\theta(a)$ and $\theta(c)$ on the boundary common to A and B. (See Fig. 2.2.)

The procedure returns $\theta(w)$ for every $a <_B w <_B c$. If $r = 1$ then there is only one such w and there are enough processors to compute $\theta(w)$ in time $O(q+\log(\rho/q))$. If $r > 1$ then all of the $\max\{1,(\rho+r)/q\}$ processors get assigned to the median of the a-to-c range and compute, for that median (call it point b), the value $\theta(b)$ in time $O(q+\log(\rho/q))$. Because of (2), it is now enough for the procedure to recursively call itself on the a-to-b range and (in parallel) the b-to-c range. The first (respectively, the second) of these recursive calls gets assigned $\max\{1,(\rho_1+r/2)/q\}$ (respectively, $\max\{1,(\rho_2+r/2)/q\}$) processors, where ρ_1 (respectively, ρ_2) is the number of points between $\theta(a)$ and $\theta(b)$ (respectively, between $\theta(b)$ and $\theta(c)$). Because $\rho_1+\rho_2=\rho$, there are enough processors available for the two recursive calls. (See Fig. 2.2.) In the initial call to the procedure, it is given (i) the whole list L_B, (ii) the θ of the first and last point of L_B, and (iii) $3m/2q$ processors. The depth of the recursion is $\log m$, at each level of which the time taken is no more than $O(q+\log(m/q))$. Therefore the procedure takes time $O((q+\log(m/q))\log m)$ with $O(m/q)$ processors. We conclude that the preliminary solution follows from (2).

We now review the proof of property (2). It is by contradiction: Suppose that, for some $w_1, w_2 \in L_B$, we have $w_1 <_B w_2$ and $\theta(w_1)$ is to the right of $\theta(w_2)$, as shown in Fig. 2.3. By definition of the function θ there is a shortest path from v to w_1 going through $\theta(w_1)$ (call this path α), and one from v to w_2 going through $\theta(w_2)$ (call it

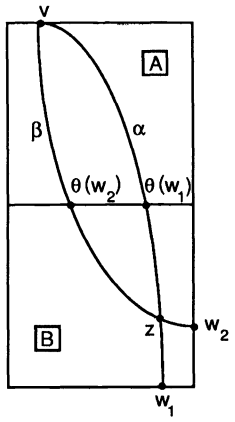


FIG. 2.3. *Illustrating the proof of property* (2).

β). Since $w_1 <_B w_2$ and $\theta(w_1)$ is to the right of $\theta(w_2)$, the two paths α and β must cross at least once somewhere in B: let z be such an intersection point. See Fig. 2.3. Let $prefix(\alpha)$ (respectively, $prefix(\beta)$) be the portion of α (respectively, β) that goes from v to z. We obtain a contradiction in each of two possible cases:

Case 1. The length of $prefix(\alpha)$ differs from that of $prefix(\beta)$. Without loss of generality, assume it is the length of $prefix(\beta)$ that is the smaller of the two. But then, the v-to-w_1 path obtained from α by replacing $prefix(\alpha)$ by $prefix(\beta)$ is shorter than α, a contradiction.

Case 2. The length of $prefix(\alpha)$ is same as that of $prefix(\beta)$. In α, replacing $prefix(\alpha)$ by $prefix(\beta)$ yields another shortest path between v and w_1, one that crosses the boundary common to A and B at a point to the left of $\theta(w_1)$, contradicting the definition of the function θ.

This completes the review of the proof of (2).

3. Using fewer processors. This section gives an algorithm that has the same time complexity as that of the previous section, but whose processor complexity is a factor of $\log m$ better. This is more than a mere "warm-up" for our best CREW algorithm of §6: the algorithm of §6 will actually use the technical result, given in this section, that $DIST_{A \cup B}$ can be obtained from $DIST_A$ and $DIST_B$ with $O(m^2)$ total work.

We establish the following lemma.

LEMMA 1. *Let G be an $m \times m$ grid* DAG. *Let $DIST_G$ be a $(2m) \times (2m)$ matrix containing the lengths of all shortest paths that begin at the top or left boundary of G, and end at the right or bottom boundary of G. The matrix $DIST_G$ can be computed in $O(\log^3 m)$ time, $O(m^2)$ space, and with $O(m^2/\log^2 m)$ processors by a* CREW-PRAM.

We prove the above lemma by giving an algorithm whose processor complexity is a $\log m$ factor better than that of the preliminary solution of §2. We illustrate the method by showing how $DIST_{A \cup B}$ can be obtained from $DIST_A$ and $DIST_B$ in $O(\log^2 m)$ time and $O(m^2/\log^2 m)$ processors. The preliminary procedure for computing $DIST_{A \cup B}$ can be seen to do a total amount of work which is $O(m^2 \log m)$. Our strategy will be to first give a procedure which has the same time and processor complexities as the preliminary one, but which does a total amount of work which

is only $O(m^2)$. Our claimed bounds for the computation of $DIST_{A\cup B}$ from $DIST_A$ and $DIST_B$ will then follow from this improved procedure and from Brent's theorem [7] as follows.

THEOREM 1 (Brent). *Any synchronous parallel algorithm taking time T that consists of a total of W operations can be simulated by P processors in time $O((W/P)+T)$.*

Proof. See [7] for the proof. □

There are actually two qualifications to Brent's theorem before we can apply it to a PRAM: (i) at the beginning of the ith parallel step, we must be able to compute the amount of work W_i done by that step, in time $O(W_i/P)$ and with P processors, and (ii) we must know how to assign each processor to its task. Both (i) and (ii) will trivially hold in our framework.

Let L_A and $<_A$ be defined analogously to L_B and $<_B$, respectively. In other words, L_A is a list of the m points on the left and top boundaries of A, sorted in the order in which they are encountered by a walk that starts at the lowest point of the left boundary of A and ends at the rightmost point of the top boundary of A (i.e., sorted by increasing order according to the $<_A$ relationship). A symmetric version of (2) holds, i.e., for any $w \in L_B$ and any two points v_1 and v_2 of L_A, we have the following:

(3) If $v_1 <_A v_2$ then $\theta(v_1, w)$ is not to the right of $\theta(v_2, w)$.

The proof of (3) is identical to that of (2) and is therefore omitted.

Let P be the $m \times (m/2)$ submatrix of $DIST_A$ containing the lengths of the shortest paths that begin at the top or left boundary of A, and end at its bottom boundary. Let Q be the $(m/2) \times m$ submatrix of $DIST_B$ containing the lengths of the shortest paths that begin at the top boundary of B, and end at its bottom or right boundary. By definition, the rows of P are indexed by the entries of L_A, the columns of Q are indexed by the entries of L_B, and the columns of P (hence the rows of Q) are indexed by the $m/2$ points at the common boundary of A and B, sorted from left to right. The problem we face is that of "multiplying" the $m \times (m/2)$ matrix P and the $(m/2) \times m$ matrix Q in the closed semiring $(\min, +)$. In matrix terminology, $\theta(v, w)$ is the smallest index k, $1 \le k \le m/2$, such that $PQ(v, w) = P(v, k) + Q(k, w)$. We give the procedure below for the (more general) case where P is an $\ell \times h$ matrix, and Q is an $h \times \ell$ matrix, $\ell \le 2h$. The only structure of these matrices that our algorithm uses is the following property (4), which is merely a restatement of properties (2) and (3) using matrix terminology:

(4) $\forall (1 \le v_1 < v_2 \le \ell, 1 \le w \le \ell)$, $\theta(v_1, w) \le \theta(v_2, w)$, and $\theta(w, v_1) \le \theta(w, v_2)$.

To compute the product of P and Q in the closed semiring $(\min, +)$, it suffices to compute $\theta(v, w)$ for all $1 \le v, w \le \ell$. To compute the product PQ (i.e., the function θ), we use the following procedure which runs in $O(\log \ell \log h)$ time, $O(\ell h / \log h)$ processors, and $O(\ell h)$ total work:

1) Recursively solve the problem for the product $P'Q'$ where P' (respectively, Q') is the $(\ell/2) \times h$ (respectively, $h \times (\ell/2)$) matrix consisting of the odd rows (respectively, odd columns) of P (respectively, Q). This gives $\theta(v, w)$ for all pairs (v, w) whose respective parities are (odd, odd). If $Work(\ell, h)$ and $T(\ell, h)$ denote the total work and time for this procedure, then this step does $Work(\ell/2, h)$ work in $T(\ell/2, h)$ time.

2) Compute $\theta(v,w)$ for all pairs (v,w) of parities (even, odd). This is done as follows. In parallel for each odd w, assign $h/\log h$ processors to w, with the task of computing $\theta(v,w)$ for all even v. The fact that we already know $\theta(v,w)$ for all odd v, together with property (4), implies that these $h/\log h$ processors are enough to do the job in $O(\log h)$ time. The work done is then $O(h)$ for each such w, for a total of $O(\ell h)$ work for this step.
3) Compute $\theta(v,w)$ for all pairs (v,w) of parities (odd, even). The method used is identical to that of the previous step and is therefore omitted.
4) Compute $\theta(v,w)$ for all pairs (v,w) of parities (even, even). The method is very similar to that of the previous two steps and is therefore omitted.

The time, processor, and work complexities of the above method satisfy the recurrences:

$$T(\ell,h) \le T(\ell/2,h) + c_1 \log h,$$

$$P(\ell,h) \le \max\{P(\ell/2,h), \ell h/\log h\},$$

$$Work(\ell,h) \le Work(\ell/2,h) + c_2 \ell h,$$

where c_1 and c_2 are constants. These recurrences imply that $T(\ell,h) = O(\log \ell \log h)$, $P(\ell,h) = O(\ell h/\log h)$, and $Work(\ell,h) = O(\ell h)$. This, together with Theorem 1 (Brent's theorem) in which $T = \log \ell \log h$, $P = \ell h/q$, and $W = \ell h$, implies that the above algorithm can be simulated by $\ell h/q$ processors in $O(q + \log \ell \log h)$ time. In our case, we have $\ell = m$ and $h = m/2$, implying that PQ (and hence $DIST_{A \cup B}$) can be obtained from P and Q in $O(q + \log^2 m)$ time with $O(m^2/q)$ processors.

The above method enables us to obtain $DIST_G$ from $DIST_A$, $DIST_B$, $DIST_C$, $DIST_D$ in $O(q + \log^2 m)$ time and $O(m^2/q)$ processors. This implies that the overall divide-and-conquer algorithm runs in $O((q + \log^2 m)\log m)$ time with $O(m^2/q)$ processors. Choosing $q = \log^2 m$ establishes Lemma 1.

4. The case $m \le n$. This section generalizes the algorithm for the case $m \le n$. The main result is the following.

THEOREM 2. *Let G be an $m \times n$ grid* DAG, $m \le n$. *The length of a shortest source-to-sink path in G can be computed by a* CREW-PRAM *in $O(\log n \log^2 m)$ time, $O(mn)$ space, and with $O(mn/\log^2 m)$ processors.*

Note that, if G is $m \times n$ with $m \le n$, then using the same idea as in §3 would result in an unacceptable $(m+n)(m+n)/\log^2(m+n)$ processor complexity, the $DIST_G$ matrix we are computing now being $(m+n) \times (m+n)$. In order to prove our claimed bounds, we shall abandon the goal of computing such a matrix $DIST_G$ and settle for computing a D_G matrix that contains less information than $DIST_G$, but enough to obtain the desired quantity: the length of a shortest source-to-sink path in G.

DEFINITION 3. For any $m \times n$ grid DAG G, $m \le n$, let D_G be the $m \times m$ matrix containing the lengths of all the shortest paths that begin at the left boundary of G, and end at the right boundary of G.

Note that D_G is a submatrix of $DIST_G$.

The following lemma is another ingredient that we need.

LEMMA 2. *Let G be an $m \times m'$ grid* DAG *that is partitioned by a vertical line into G_1 and G_2. (See Fig.* 4.1.*) Then, given D_{G_1} and D_{G_2}, the matrix D_G can be computed by a* CREW-PRAM *in $O(\log^2 m)$ time, $O(m^2)$ space, and with $O(m^2/\log^2 m)$ processors.*

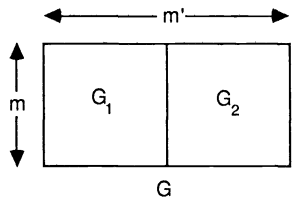


FIG. 4.1. *Illustrating Lemma* 2.

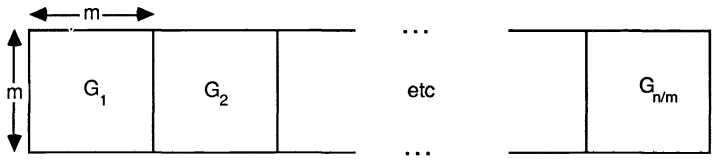


FIG. 4.2. *Illustrating the partitioning of G.*

Proof. The algorithm proving the above lemma is similar to the procedure we used in §3 to obtain $DIST_{A \cup B}$ from $DIST_A$ and $DIST_B$, and is omitted. □

We are now ready to prove Theorem 2.

Proof of Theorem 2. Without loss of generality, assume that m divides n (if not then G can always be "padded" with extra vertices and zero-cost edges so as to make it $m \times n'$ where m divides n' and $n' - n \leq m$). Partition G by vertical lines into n/m grid DAGs $G_1, \cdots, G_{n/m}$, where each G_i is $m \times m$ (see Fig. 4.2). In parallel for each $i \in \{1, \cdots, n/m\}$, use Lemma 1 to obtain the $DIST_{G_i}$ matrices. This takes $O(\log^3 m)$ time with a total of $O((m^2/\log^2 m)(n/m)) = O(mn/\log^2 m)$ processors. From each $DIST_{G_i}$ matrix, extract its submatrix D_{G_i}. We are now left with the task of combining the D_{G_i}'s into a single D_G. In parallel, we recursively obtain the D-matrix of the union of the leftmost $n/2m$ G_i's, and similarly the D-matrix of the union of the rightmost $n/2m$ G_i's. We then combine these two D matrices into D_G by using Lemma 2. This recursive combining procedure takes a total of $O(\log^2 m \log(n/m))$ time with $O(mn/\log^2 m)$ processors. The overall time complexity is therefore $O(\log^3 m + \log^2 m \log(n/m)) = O(\log n \log^2 m)$. □

In view of the remarks made in §1, the following is an immediate consequence of the above theorem.

COROLLARY 1. *Let x and y be two strings over an alphabet I. Let $m = \min(|x|, |y|)$, $n = \max(|x|, |y|)$. For edit operations of arbitrary nonnegative costs, the edit distance from x to y can be computed by a* CREW-PRAM *in $O(\log n \log^2 m)$ time, $O(mn)$ space, and with $O(mn/\log^2 m)$ processors.*

5. Computing the actual path. In this section we sketch a modification of the algorithm given in the previous sections which enables us to compute an actual shortest source-to-sink path in G within the same time, space, and processor bounds as in the length computation.

THEOREM 3. *Let G be an $m \times n$ grid* DAG, *$m \leq n$. A shortest source-to-sink path in G can be computed by a* CREW-PRAM *in $O(\log n \log^2 m)$ time, $O(mn)$ space, and with $O(mn/\log^2 m)$ processors.*

The rest of this section proves the above theorem.

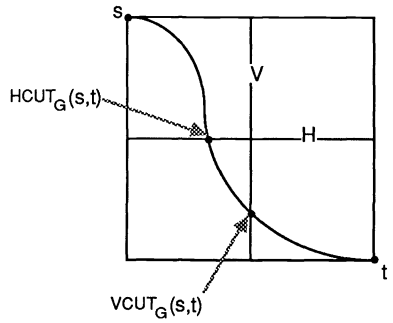


FIG. 5.1. *Illustrating the computation of the actual path.*

We begin with the case $m = n$, i.e., an $m \times m$ grid DAG. We cannot afford to let the matrix $DIST_G$ of §3 be a matrix of paths instead of lengths, because that would take m^3 space, killing any hope of a polylog time algorithm that does not use an almost cubic number of processors. Instead, we modify the algorithm of §3 so that it also has the "side effect" of computing two $(2m) \times (2m)$ matrices $HCUT_G$ and $VCUT_G$ (mnemonics for "horizontal cut" and "vertical cut," respectively) having the same index domain as $DIST_G$. These two matrices are *global* in the sense that they remain even after the recursive call returns, and their significance is as follows. Let H be the horizontal boundary between $A \cup C$ and $B \cup D$, and let V be the vertical boundary between $A \cup B$ and $C \cup D$ (see Fig. 5.1). Let $PATH(x, y)$ be the *lowest* x-to-y path of cost $DIST_G(x, y)$; i.e., no other x-to-y path of length $DIST_G(x, y)$ goes through any vertex that is below a vertex of $PATH(x, y)$. It is easy to prove that there is a unique such path $PATH(x, y)$ (the proof is straightforward and is omitted). Then $HCUT_G(x, y)$ is the leftmost intersection of $PATH(x, y)$ with H, and $VCUT_G(x, y)$ is the lowest intersection of $PATH(x, y)$ with V. If the intersection of $PATH(x, y)$ with H (respectively, V) is empty, then $HCUT_G(x, y)$ (respectively, $VCUT_G(x, y)$) is undefined. Because these additional matrices are global, after the algorithm terminates it leaves behind $N(m)$ of them where

$$N(m) = 4N(m/2) + 2 = O(m^2).$$

Fortunately, even though there are $O(m^2)$ such $HCUT$ and $VCUT$ matrices that remain, the total storage space they take is $S(m)$ where

$$S(m) = 4S(m/2) + cm^2 = O(m^2 \log m).$$

Before showing how $S(m)$ is decreased to $O(m^2)$, we show how the matrices $HCUT$ and $VCUT$ are used to retrieve the shortest source-to-sink path in G. It suffices to output the points on this path as a set (i.e., in arbitrary order), since a postprocessing sorting step puts them in the right order in $O(\log m)$ time and $O(m)$ processors [8]. Let s and t denote the source and sink of G, respectively. We first print $HCUT_G(s, t)$ and $VCUT_G(s, t)$, and then we recursively print the three portions of the shortest s-to-t path determined by its two intersections with H and V (this involves three $(m/2) \times (m/2)$ grid DAGs; see Fig. 5.1). The procedure can be implemented to run in $O(h + \log m)$ time and $2m/h$ processors, where $h \leq m$ is an integer of our choice, by maintaining the property that each recursive call of size $m' \geq h$ gets assigned

$2m'/h$ processors (the bottom of the recursion is when problem size m' becomes $\leq h$, at which time a single processor finishes the job sequentially, in $O(m')$ time). (We would, of course, choose $h = \log m$.)

We bring the space complexity $S(m)$ down by storing each row (say, row p) of the $HCUT$ (or $VCUT$) matrix in an $O(m)$-bit vector $ROW(p)$ that is "packed" in $O(m/\log m)$ registers of size $\log m$ bits each. (The assumption that word size is a logarithmic function of problem size is a standard one [5].) Let us immediately point out that a consequence of this encoding scheme is that we now have $S(m) = O(m^2)$. To see this, let $BITS(m)$ be the total number of bits used by the encoding scheme, and note that $S(m) = O(BITS(m)/\log m)$, since each register contains $\log m$ bits. Thus it suffices to show that $BITS(m) = O(m^2 \log m)$. But this trivially follows from the fact that $BITS(m) = 4BITS(m/2) + O(m^2)$.

We now describe the encoding scheme used for storing row p of (e.g.) $HCUT$ in the $O(m)$-bit vector $ROW(p)$. We exploit the fact that the contents of row p happen to be sorted by the left-to-right linear ordering of the points on H. More precisely, if the points of H are denoted by $1, \cdots, m$ in left-to-right order, then row p contains a nondecreasing sequence of $O(m)$ integers between 1 and m. Instead of storing the entries of row p, we therefore store the sequence of *differences* between the consecutive entries of row p. This sequence of differences is stored in unary in the $O(m)$-bit vector $ROW(p)$, with as many consecutive 1's as needed to encode a particular difference, and using a 0 as a separator between consecutive nonzero entries. For example, if row p contains the sequence $(3, 3, 5, 7, 9, 11)$ then the sequence of differences is $(3, 0, 2, 2, 2, 2)$ and $ROW(p) = (11100110110110110)$. We can actually obtain $ROW(p)$ without going through the intermediate step of computing the sequence of differences: simply observe that if the ith entry of row p is k then the $(i + k)$th entry of $ROW(p)$ is a 0 (in our example, the fourth entry is 7 and hence the eleventh entry of $ROW(p)$ is a 0). This observation implies that we can obtain $ROW(p)$ in $O(q + \log m)$ time with $O(m/q)$ processors by first initializing all the entries of $ROW(p)$ to 1, and then changing some of these into 0's according to the observation. Reading the kth entry of row p is now done by computing the sum of all the entries of $ROW(p)$ that precede its kth leftmost zero; i.e. it requires a parallel prefix computation [11] on $ROW(p)$ and hence $O(\log m)$ time, so that extracting the s-to-t path now takes $O(\log^2 m)$ time rather than the previous $O(\log m)$. This fact is of no consequence, however, since the bottleneck in the time complexity comes from the computation of the $DIST_G$ matrix.

This completes the proof of Theorem 3 for the case $m = n$.

It is not hard to see that, so long as $m = n$, the above procedure actually works when s and t are arbitrary points on the boundary of G. This observation implies that, for the case $m \leq n$, it suffices to find for every $i \in \{1, \cdots, (n/m) - 1\}$ the lowest point (call it $CROSS(i)$) at which a shortest path from s to t crosses the boundary between G_i and G_{i+1}. Once we have these $CROSS(i)$'s, we can use the procedure of the previous paragraph to obtain the actual path joining each $CROSS(i)$ to $CROSS(i+1)$ in time $O(\log^3 m)$, space $O(m^2 n/m) = O(mn)$, and with $O((m^2/\log^2 m)(n/m)) = O(mn/\log^2 m)$ processors. We obtain the $CROSS(i)$'s as follows. Refer to §4, the proof of Theorem 2: We modify that procedure so that, as the procedure computes the D-matrix, it now also produces as a side effect a global $m \times m$ matrix CUT_G. The significance of this matrix is that $CUT_G(x, y)$ is the lowest point of intersection of any shortest x-to-y path with the boundary separating the two recursive calls. The total number of such CUT matrices is $O(n/m)$, and their total storage is $O(mn)$. We use these CUT matrices to output the $CROSS(i)$'s as a set (i.e., unordered) by first

printing $CUT_G(s,t)$, and then recursively printing the $CROSS(i)$'s that are to the left of $CUT_G(s,t)$, and simultaneously (i.e., in parallel) those to its right. It is easily seen that the $CROSS(i)$'s are produced in time $O(\log(n/m))$, and that there are enough processors to carry out the procedure. A post-processing sorting step orders the $CROSS(i)$'s. This completes the proof of Theorem 3. □

An immediate consequence of Theorem 3 is the following.

COROLLARY 2. *Let x and y be two strings over an alphabet I. Let $m = \min(|x|,|y|)$, $n = \max(|x|,|y|)$. For edit operations of arbitrary nonnegative costs, an optimal edit script from x to y can be computed by a* CREW-PRAM *in $O(\log n \log^2 m)$ time, $O(mn)$ space, and with $O(mn/\log^2 m)$ processors.*

6. A faster CREW-PRAM algorithm. This section gives a CREW algorithm that is faster by a $\log m$ factor and uses $O(mn/\log m)$ processors. More precisely, we establish the following.

THEOREM 4. *Let G be an $m \times n$ grid* DAG, *$m \leq n$. A shortest source-to-sink path in G can be computed by a* CREW-PRAM *in $O(\log n \log m)$ time, $O(mn)$ space, and with $O(mn/\log m)$ processors.*

COROLLARY 3. *Let x and y be two strings over an alphabet I. Let $m = \min(|x|,|y|)$, $n = \max(|x|,|y|)$. For edit operations of arbitrary nonnegative costs, an optimal edit script from x to y can be computed by a* CREW-PRAM *in $O(\log n \log m)$ time, $O(mn)$ space, and with $O(mn/\log m)$ processors.*

From the developments of §§2–5, it should be clear that in order to establish the above theorem, it suffices to show that:

1) The matrix $DIST_{A\cup B}$ can be obtained from $DIST_A$ and $DIST_B$ in $O(\log m)$ time, $O(m^2)$ space, and with $O(m^2/\log m)$ processors, and
2) The matrix D_G can be obtained from D_{G_1} and D_{G_2} (see Definition 3 and Fig. 4.1) in $O(\log m)$ time, $O(m^2)$ space, and with $O(m^2/\log m)$ processors.

Since the proofs of 1) and 2) are very similar, we only give that for 1). Thus the rest of this section deals with how to obtain $DIST_{A\cup B}$ from $DIST_A$ and $DIST_B$ in $O(\log m)$ time, $O(m^2)$ space, and with $O(m^2/\log m)$ processors.

6.1. Obtaining one row of $DIST_{A\cup B}$. This section gives an $O(\log m)$ time, $O(m \log m)$ space, and $O(m \log m)$ processor algorithm for obtaining one particular row of $DIST_{A\cup B}$, i.e., computing $\theta(v,w)$ for a fixed $v \in L_A$ and all $w \in L_B$. The fixed vertex v is implicit in the rest of this section, so that whenever we refer to a "path to w" it is understood that this path originates at v. To simplify the exposition, we assume that m is a power of 2 (the procedure can easily be modified for the general case).

We refer to the vertices on the boundary common to A and B (denoted $A \cap B$ for short) as *crossing vertices* and number them $c_1, c_2, \cdots, c_{m/2}$, where the numbering is from left to right along the common boundary. We refer to the vertices in L_B as *destination vertices* and denote them w_1, w_2, $\cdots$, w_m, numbered according to $<_B$, their order in L_B.

DEFINITION 4. A *crossing interval* is a nonempty set of contiguous crossing vertices $\{c_i, c_{i+1}, \cdots, c_j\}$. We say that crossing interval I is *to the left of* crossing interval J, and J is *to the right of* I, if the rightmost vertex of I is to the left of the leftmost vertex of J.

DEFINITION 5. Let $F \subseteq A \cap B$ and $w \in L_B$, i.e., F is a set of crossing vertices (not necessarily an interval) and w is a destination vertex. Let $\theta_F(w)$ denote the leftmost crossing vertex in F incident to a (v,w) path that is shortest among all (v,w) paths

constrained to pass through F. (If there is no such (v, w) path, then this is denoted by $\theta_F(w) = \emptyset$.)

Note that $\theta_F(w)$ may differ from $\theta(v, w)$, but that $\theta_{A\cap B}(w) = \theta(v, w)$.

The following lemma is the analogue, for constrained paths, to property (2) of §2.

LEMMA 3. *Let $F \subseteq A \cap B$ and $w_1, w_2 \in L_B$. If $w_1 <_B w_2$, then $\theta_F(w_1)$ is not to the right of $\theta_F(w_2)$.*

Proof. The proof is identical to that of property (2). □

We now give an informal description of the algorithm.

If U is any set of destination vertices and I is any crossing interval, then we will define $\theta_I(U)$ to be a data structure that contains enough information to determine $\theta_I(w)$ for all $w \in U$. The details of that data structure will be explained later.

It is useful to think of the computation as progressing through the nodes of a tree T which we now proceed to define.

We define a crossing interval to be *diadic* if it is either $A \cap B$ (i.e., it consists of all crossing vertices), or if it is the the left or right half of a diadic crossing interval. Note that there are exactly $m - 1$ diadic crossing intervals, which form a complete binary tree T rooted at $A \cap B$, and whose $m/2$ leaves are the $m/2$ crossing vertices (the ith leaf of T containing c_i, the ith leftmost crossing vertex). Thus the diadic crossing interval at an interior node of T is simply the union of the diadic crossing intervals of its two children in T. We can talk about the *height* and the *children* of a diadic crossing interval (= its height and children in T).

Since the $m - 1$ diadic crossing intervals are the only crossing intervals we shall be interested in, from now on we simply say "interval" as a shorthand for "diadic crossing interval." Thus whenever we refer to an interval I we are implicitly assuming that $I \in T$, i.e., that I is one of the $m - 1$ diadic crossing intervals. We use $|I|$ to denote the size of the interval, i.e., the number of crossing vertices in it. Observe that $\sum_{I\in T} |I| = O(m \log m)$. Thus we have enough processors to associate $|I|$ of them with each interval I (i.e., node I) of T. Similarly, we can afford to use $O(|I|)$ space per interval I. The computation proceeds in $2 \log m - 1$ *stages*, each of which takes constant time. The ultimate goal is for every interval I to compute $\theta_I(L_B)$. The structure of the algorithm is reminiscent of the *cascading divide-and-conquer* technique [8], [6]: each $I \in T$ will compute $\theta_I(U)$ for progressively larger subsets U of L_B, subsets U that double in size from one stage to the next of the computation. We now proceed to state precisely what these subsets are.

DEFINITION 6. A *k-sample* of L_B is obtained by choosing every kth element of L_B (i.e., every element whose rank in L_B is a multiple of k). For example, a 4-sample of L_B is $(w_4, w_8, \cdots, w_m)$. For $k \in \{0, 1, \cdots, \log m\}$, let U_k denote an $(m/2^k)$-sample of L_B.

For example:

$$
\begin{aligned}
U_0 &= \{w_m\},\\
U_1 &= \{w_{m/2}, w_m\},\\
U_2 &= \{w_{m/4}, w_{m/2}, w_{3m/4}, w_m\},\\
U_3 &= \{w_{m/8}, w_{m/4}, w_{3m/8}, w_{m/2}, w_{5m/8}, w_{3m/4}, w_{7m/8}, w_m\},\\
&\cdots\\
U_{\log m} &= \{w_1, w_2, \cdots, w_m\} = L_B.
\end{aligned}
$$

Note that $|U_k| = 2^k = 2|U_{k-1}|$.

At the tth stage of the algorithm, an interval I of height h in T will use its $|I|$ processors to compute, in constant time, $\theta_I(U_{t-h})$ if $h \le t \le h + \log m$. It does so with

the help of information from $\theta_I(U_{t-1-h})$, $\theta_{\text{LeftChild}(I)}(U_{t-h})$, and $\theta_{\text{RightChild}(I)}(U_{t-h})$, all of which are available from the previous stage $t-1$. If $h > t$ or $t > h + \log m$ then interval I does nothing during stage t. Thus before stage h the interval I lies "dormant," then at stage $t = h$ it first "wakes up" and computes $\theta_I(U_0)$, then at the next stage $t = h+1$ it computes $\theta_I(U_1)$, etc. At step $t = h + \log m$ it computes $\theta_I(U_{\log m})$, after which it is done. The details of what information I stores and how it uses its $|I|$ processors to perform stage t in constant time are given below. First, we observe the following.

LEMMA 4. *The algorithm terminates after* $2\log m - 1$ *stages.*

Proof. After stage $h + \log m$ every interval I of height h is done, i.e., it has computed $\theta_I(L_B)$. The root interval has height $\log m - 1$ and thus is done after stage $2\log m - 1$. □

Thus to establish the main claim of this section, it suffices to prove the following lemma.

LEMMA 5. *With* $|I|$ *processors and* $O(|I|)$ *space assigned to each interval* $I \in T$, *every stage of the algorithm can be completed in constant time.*

The rest of this section proves the above lemma.

We begin by describing the way in which an interval I at height h in T stores $\theta_I(U_{t-h})$ using only $|I|$ space. Rather than directly storing the values $\theta_I(w)$ for all $w \in U_{t-h}$ (which would require $|U_{t-h}|$ space), we store instead the *inverse* mapping, which turns out to have a compact $O(|I|)$ space encoding because of the monotonicity property guaranteed by Lemma 3. In other words, for each $c \in I$, let

$$\pi_I(c,t) = \{w \in U_{t-h} | \theta_I(w) = c\}.$$

Then Lemma 3 implies that the elements of $\pi_I(c,t)$ are contiguous in the list U_{t-h}. More specifically, the sets $\pi_I(c,t)$, $c \in I$, form a partition of the set U_{t-h} into $|I|$ subsets each of which is either empty or contains contiguous elements in U_{t-h}. Therefore I does not need to store the elements of $\pi_I(c,t)$ explicitly, but rather by just remembering where they begin and end in U_{t-h}, i.e., $O(1)$ space for each $c \in I$. Of course U_{t-h} is itself not stored explicitly by I, since the height h and stage number t implicitly determine it. Thus $O(|I|)$ space is enough for storing $\pi_I(c,t)$ for all $c \in I$.

Interval I stores the sets $\pi_I(c,t)$, $c \in I$, in an array $RANGE_I$, with entries $RANGE_I(c) = (w_i, w_j)$ such that w_i (respectively, w_j) is the first (respectively, last) element of U_{t-h} that belongs to $\pi_I(c,t)$. If $\pi_I(c,t)$ is empty then $RANGE_I(c)$ equals $\emptyset$. At stage t of the algorithm, I must update the $RANGE_I$ array so that it changes from being a description of the $\pi_I(c,t-1)$'s to being a description of the $\pi_I(c,t)$'s. The rest of this section needs only to show how such an update is done in constant time by the $|I|$ processors assigned to I. Of course, since we are ultimately interested in $\theta_{A\cap B}(w)$ for every $w \in L_B$, at the end of the algorithm we must run a postprocessing procedure which recovers this information from the $RANGE_{A\cap B}$ array available at the root of T, i.e., it explicitly obtains $\theta_{A\cap B}(w)$ for all $w \in U_{\log m}$. But this postprocessing is trivial to perform in $O(\log m)$ time with $O(m)$ processors, and we shall not concern ourselves with it anymore.

In the rest of this section, intervals L and R are the left and (respectively) right children of I in T. Observe that, for any destination w, $\theta_I(w)$ is one of $\theta_L(w)$ or $\theta_R(w)$. Furthermore, if $\theta_I(w) = \theta_L(w)$ then $\theta_I(w') \in L$ for every w' smaller than w (in the $<_B$ ordering). Similarly, if $\theta_I(w) = \theta_R(w)$ then $\theta_I(w') \in R$ for any w' larger than w. (These observations follow from Lemma 3.)

The $RANGE_I$ array alone is not enough to enable I to perform the updating

required at stage t. In addition, at each stage t, I must compute in a register called $CRITICAL_I$ an entry $Critical_I(t)$ defined as follows.

DEFINITION 7. At each stage t, let the *critical destination* for I, denoted $Critical_I(t)$, be the largest $w \in U_{t-h}$ such that $\theta_I(w) = \theta_L(w)$. If there is no such w (i.e., if $\theta_I(w) = \theta_R(w)$ for all $w \in U_{t-h}$), then $Critical_I(t) = \emptyset$.

Note that Lemma 3 ensures that $Critical_I(t)$ is well defined. We shall later show how storing and maintaining this critical destination enables I to update the $RANGE_I$ array in constant time. Of course it also places on I the burden of updating its $CRITICAL_I$ register so that after stage t it contains $Critical_I(t)$ rather than $Critical_I(t-1)$. We shall later show that updating the $CRITICAL_I$ register can be done in constant time as well.

We now complete this section by explaining how I performs stage t, i.e., how it obtains $Critical_I(t)$ and the $\pi_I(c,t)$'s using the $\pi_L(c,t-1)$'s, the $\pi_R(c,t-1)$'s, and its previous critical index $Critical_I(t-1)$. The fact that the $|I|$ processors can do this in constant time is based on the following three observations:

(5) $Critical_I(t)$ is either the same as $Critical_I(t-1)$, or it is the successor of $Critical_I(t-1)$ in U_{t-h}.

(6) If $c \in L$ then $\pi_I(c,t) = \pi_L(c,t-1) - \{$the elements of $\pi_L(c,t-1)$ that are larger than $Critical_I(t)$ in the $<_B$ ordering$\}$.

(7) If $c \in R$ then $\pi_I(c,t) = \pi_R(c,t-1) - \{$the elements of $\pi_R(c,t-1)$ that are less than or equal to $Critical_I(t)$ in the $<_B$ ordering$\}$.

Correctness of (5)–(7) follows from the definitions. Their algorithmic implications are discussed next.

Updating the $CRITICAL_I$ register. Relationship (5) implies that in order to update $CRITICAL_I$ (i.e., compute $Critical_I(t)$) all I has to do is determine which of $Critical_I(t-1)$ or its successor in U_{t-h} is the correct value of $Critical_I(t)$. This is done as follows. If $Critical_I(t-1)$ has no successor in U_{t-h} then $Critical_I(t-1) = w_m$ and hence $Critical_I(t) = Critical_I(t-1)$. Otherwise the updating is done in the following two steps. For shorthand, let r denote $Critical_I(t-1)$, and let s denote the successor of r in U_{t-h}.

- The first step is to compute $\theta_L(s)$ and $\theta_R(s)$ in constant time. This involves a search in L (respectively, R) for the crossover c in L (respectively, R) whose $\pi_L(c,t-1)$ (respectively, $\pi_R(c,t-1)$) contains s. These two searches in L and R are done in constant time with the $|I|$ processors available. We explain how the search in L is done (that in R is similar and omitted). I assigns a processor to each $c \in L$, and that processor tests whether s is in $\pi_L(c,t-1)$; the answer is "yes" for exactly one of those $|L|$ processors and thus can be collected in constant time. Thus I can determine $\theta_L(s)$ and $\theta_R(s)$ in constant time.
- The next step consists of comparing which of the following two paths to s is better: the one through $\theta_L(s)$, or the one through $\theta_R(s)$. If the path through $\theta_R(s)$ is better, then $Critical_I(t)$ is the same as $Critical_I(t-1)$ and the $CRITICAL_I$ register stays the same (containing r). Otherwise $Critical_I(t)$ is s, and we set $CRITICAL_I$ equal to s. This comparison of the two paths and resulting update are done in constant time (by one processor, in fact).

We next show how the just computed $Critical_I(t)$ value is used to compute the $\pi_I(c,t)$'s in constant time.

Updating the $RANGE_I$ array. Relationship (6) implies the following for each $c \in L$:

1) If $\pi_L(c, t-1)$ is to the left of $Critical_I(t)$ then $\pi_I(c,t) = \pi_L(c, t-1)$.
2) If $\pi_L(c, t-1)$ is to the right of $Critical_I(t)$ then $\pi_I(c,t) = \emptyset$.
3) If $\pi_L(c, t-1)$ contains $Critical_I(t)$ then it consists of the portion of $\pi_L(c, t-1)$ up to (and including) $Critical_I(t)$.

The above facts 1)–3) immediately imply that $O(1)$ time is enough for $|L|$ of the $|I|$ processors assigned to I to compute $\pi_I(c,t)$ for all $c \in L$, by adjusting the $RANGE_I(c)$ value according to rules 1)–3) above (recall that the $\pi_L(c, t-1)$'s are available in L from the previous stage $t-1$, and $Critical_I(t)$ has already been computed and is in the $CRITICAL_I$ register).

A similar argument shows that relationship (7) implies that $|R|$ processors are enough for computing $\pi_I(c,t)$ for all $c \in R$. Thus I can update its $RANGE_I$ array in constant time with $|I|$ processors. This completes the proof of Lemma 5.

The result of this section is easily seen to provide an $O(\log m)$ time, $O(m \log m)$ processor CREW-PRAM solution to the problem commonly called [2], [3] "computing the row maxima of an $m \times m$ totally monotone matrix" (we refer the reader to [2] and [3] for some of the many applications of this problem, for which a linear-time sequential solution is known [2]).

6.2. Obtaining all rows of $DIST_{A \cup B}$. This section shows that $O(m^2/\log m)$ processors and $O(m^2)$ space suffice for computing in $O(\log m)$ time all the $\theta(v,w)$'s (hence for computing the $DIST_{A \cup B}$ matrix). Let L_A and L_B be as in previous sections. Our task is to compute $\theta(v,w)$ for all $v \in L_A$ and all $w \in L_B$. We use $S(L,k)$ to denote the k-sample of a list L.

In the first stage of the computation, we assign $m \log m$ processors to each $v \in S(L_A, \log^2 m)$. Then, in parallel for all $v \in S(L_A, \log^2 m)$, we use the method of the previous section to obtain $\theta(v,w)$ for all $w \in L_B$. This first stage of the computation takes $O(\log m)$ time, $O(m^2)$ space, and $O(m^2/\log m)$ processors, and obtains $\theta(v,w)$ for all $v \in S(L_A, \log^2 m)$ and $w \in L_B$.

In the second stage of the computation, we assign $2m$ processors to each $w \in S(L_B, \log m)$, with the task of computing $\theta(v,w)$ for all $v \in L_A$. These $2m$ processors perform this computation for their particular w in $O(\log m)$ time, as follows. The set of $m/\log^2 m$ values $\{\theta(v,w) \mid v \in S(L_A, \log^2 m)\}$ partitions the common boundary of A and B into $m/\log^2 m$ pieces $J_1, J_2, \cdots$ (see Fig. 6.1). Let $I_1, I_2, \cdots$ be the $m/\log^2 m$ pieces (of size $\log^2 m$ each) into which $S(L_A, \log^2 m)$ partitions L_A (see Fig. 6.1). Partition the group of $2m$ processors assigned to w into $m/\log^2 m$ subgroups, where the ith subgroup contains $\log^2 m + |J_i|$ processors whose task is to compute, for all $v \in I_i$, which element of J_i equals $\theta(v,w)$. This subgroup of $\log^2 m + |J_i|$ processors does this as follows:

1) It gives each of the $\log m$ elements of $S(I_i, \log m)$ (say, to element v) $1 + |J_i|/\log m$ processors that v uses to find out, in $O(\log m)$ time, which element of J_i equals $\theta(v,w)$. The set of $\log m$ values $\{\theta(v,w) \mid v \in S(I_i, \log m)\}$ partitions J_i into $\log m$ pieces $J_{i,1}, J_{i,2}, \cdots$. Let $I_{i,1}, I_{i,2}, \cdots$ be the $\log m$ pieces (of size $\log m$ each) into which $S(I_i, \log m)$ partitions I_i.
2) It partitions its $\log^2 m + |J_i|$ processors into $\log m$ subsubgroups, where the kth subsubgroup contains $\log m + |J_{i,k}|$ processors whose task is to compute, for all $v \in I_{i,k}$, which element of $J_{i,k}$ equals $\theta(v,w)$. This subsubgroup of $\log m + |J_{i,k}|$ processors does this in $O(\log m)$ time by giving to each of the $\log m$ elements of $I_{i,k}$ (say, to element v) $1 + |J_{i,k}|/\log m$ processors that v uses to find out, in $O(\log m)$ time, which element of $J_{i,k}$ equals $\theta(v,w)$.

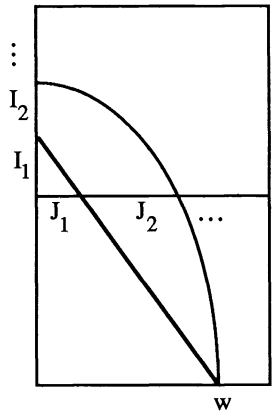


FIG. 6.1. *Illustrating the second stage of the computation.*

In the third stage of the computation, we assign $2m/\sqrt{\log m}$ processors to each $v \in S(L_A, \sqrt{\log m})$, with the task of computing $\theta(v, w)$ for all $w \in L_B$. These $2m/\sqrt{\log m}$ processors perform this computation for their particular v in $O(\log m)$ time, as follows. The set of $m/\log m$ values $\{\theta(v, w) \mid w \in S(L_B, \log m)\}$ partitions the common boundary of A and B into $m/\log m$ pieces $J_1, J_2, \cdots$. Let $I_1, I_2, \cdots$ be the $m/\log m$ pieces (of size $\log m$ each) into which $S(L_B, \log m)$ partitions L_B. Partition the group of $2m/\sqrt{\log m}$ processors assigned to v into $m/\log m$ subgroups, where the ith subgroup contains $\sqrt{\log m} + |J_i|/\sqrt{\log m}$ processors whose task is to compute, for all $w \in I_i$, which element of J_i equals $\theta(v, w)$. This subgroup of $\sqrt{\log m} + |J_i|/\sqrt{\log m}$ processors does this as follows:

1) It gives each of the $\sqrt{\log m}$ elements of $S(I_i, \sqrt{\log m})$ (say, to element w) $1 + |J_i|/\log m$ processors that w uses to find out, in $O(\log m)$ time, which element of J_i equals $\theta(v, w)$. The set of $\sqrt{\log m}$ values $\{\theta(v, w) \mid w \in S(I_i, \sqrt{\log m})\}$ partitions J_i into $\sqrt{\log m}$ pieces $J_{i,1}, J_{i,2}, \cdots$. Let $I_{i,1}, I_{i,2}, \cdots$ be the $\sqrt{\log m}$ pieces (of size $\sqrt{\log m}$ each) into which $S(I_i, \sqrt{\log m})$ partitions I_i.
2) It partitions its $\sqrt{\log m} + |J_i|/\sqrt{\log m}$ processors into $\sqrt{\log m}$ subsubgroups. The kth subsubgroup contains $1 + |J_{i,k}|/\sqrt{\log m}$ processors whose task is to compute, for all $w \in I_{i,k}$, which element of $J_{i,k}$ equals $\theta(v, w)$. This subsubgroup of $1 + |J_{i,k}|/\sqrt{\log m}$ processors does this in $O(\log m)$ time as follows:
   (a) If $|J_{i,k}| \geq \log m$, by giving to each of the $\sqrt{\log m}$ elements of $I_{i,k}$ (say, to element w) $|J_{i,k}|/\log m$ processors that w uses to find out, in $O(\log m)$ time, which element of $J_{i,k}$ equals $\theta(v, w)$.
   (b) If $|J_{i,k}| < \log m$, by partitioning $I_{i,k}$ into $1 + |J_{i,k}|/\sqrt{\log m}$ equal pieces $I_{i,k,1}, I_{i,k,2}, \cdots$ (each of size at most $\log m/|J_{i,k}|$) and giving each $I_{i,k,l}$ one processor. This processor sequentially finds $\theta(v, w)$ for all $w \in I_{i,k,l}$ in $O(\log m)$ time, since $|I_{i,k,l}||J_{i,k}| = O(\log m)$.

The fourth stage of the computation "fills in the blanks" by actually computing $\theta(v, w)$ for all $v \in L_A$ and $w \in L_B$. It does so with only $m^2/\log m$ processors by exploiting what was computed in the previous stages. Partition L_A into $m/\sqrt{\log m}$ contiguous blocks $X_1, X_2, \cdots$ of size $\sqrt{\log m}$ each. Similarly, partition L_B into $m/\sqrt{\log m}$ contiguous blocks $Y_1, Y_2, \cdots$ of size $\sqrt{\log m}$ each. Let Z_{ij} be the interval on the boundary common to A and B that is defined by the set of $\theta(v, w)$ such that $v \in X_i$ and $w \in Y_j$. Of course we already know the beginning and end of each such interval

Z_{ij} (from the third stage of the computation). Furthermore, we have the following lemma.

LEMMA 6. $\sum_{i=1}^{m/\sqrt{\log m}} \sum_{j=1}^{m/\sqrt{\log m}} |Z_{ij}| = O(m^2/\sqrt{\log m})$.

First, observe that Z_{ij} and $Z_{i+1,j+1}$ are adjacent intervals that are disjoint except for one possible common endpoint (the rightmost point in Z_{ij} and the leftmost point in $Z_{i+1,j+1}$ may coincide). This observation implies that for any given integer δ ($0 \leq |\delta| \leq m/\sqrt{\log m}$), we have (it is understood that $|Z_{ij}| = 0$ if $j < 1$ or $j > m/\sqrt{\log m}$):

$$\sum_{i=1}^{m/\sqrt{\log m}} |Z_{i,i+\delta}| = O(m).$$

The lemma follows from the above simply by rewriting the summation in the lemma's statement:

$$\sum_{\delta=-m/\sqrt{\log m}}^{m/\sqrt{\log m}} \sum_{i=1}^{m/\sqrt{\log m}} |Z_{i,i+\delta}|. \qquad \square$$

The above lemma implies that with a total of $m^2/\log m$ processors, we can afford to assign a group of $1 + |Z_{ij}|/\sqrt{\log m}$ processors to each pair X_i, Y_j. The task of this group is to compute $\theta(v,w)$ for all $v \in X_i$ and $w \in Y_j$ (of course each such $\theta(v,w)$ is in Z_{ij}). It suffices to show how such a group performs this computation in $O(\log m)$ time. If $|Z_{ij}| \leq \sqrt{\log m}$ then a single processor can solve the problem in $O((\sqrt{\log m})^2) = O(\log m)$ time, by the quadratic work method of §3. If $|Z_{ij}| > \sqrt{\log m}$ then we partition Z_{ij} into $|Z_{ij}|/\sqrt{\log m}$ pieces $J_1, J_2, \cdots$ of size $\sqrt{\log m}$ each. We assign to each J_k one processor which solves sequentially the subproblem defined by X_i, J_k, Y_j, i.e., it computes for each $v \in X_i$ and $w \in Y_j$ the leftmost point of J_k through which passes a path that is shortest among the v-to-w paths that are constrained to go through J_k. This sequential computation takes $O(\log m)$ time (again, using the method of §3). It is done in parallel for all the J_k's. Now we must, for each pair v, w with $v \in X_i$ and $w \in Y_j$, select the best crossing point for it among the $|Z_{ij}|/\sqrt{\log m}$ possibilities returned by each of the above-mentioned sequential computations. This involves a total (i.e., for all such v, w pairs) of $O(|X_i||Y_j||Z_{ij}|/\sqrt{\log m}) = O(|Z_{ij}|\sqrt{\log m})$ comparisons, which can be done in $O(\log m)$ time by the $|Z_{ij}|/\sqrt{\log m}$ processors available (Brent's theorem).

7. CRCW-PRAM algorithm. This section briefly sketches how the partitioning schemes of §6.2 translate into a CRCW-PRAM algorithm of time complexity $O(\log n(\log\log m)^2)$ and processor complexity $O(mn/\log\log m)$. Again, it suffices to show how $DIST_{A\cup B}$ can be obtained from $DIST_A$ and $DIST_B$ in $O((\log\log m)^2)$ time and with $m^2/\log\log m$ processors.

We first describe a preliminary procedure that has the right time complexity but does too much work: $O(m^2 \log m)$ work. The procedure is recursive, and we describe it for the more general case when $DIST_A$ is $\ell \times h$ and $DIST_B$ is $h \times \ell$ (that is, $|L_A| = |L_B| = \ell$ and the common boundary has size h). It suffices to show that we can, in $O(\log\log h \log\log \ell)$ time and $\ell h \log \ell$ work, compute $\theta(v,w)$ for all $v \in L_A$ and $w \in L_B$.

The first stage of the preliminary algorithm partitions L_A into $\sqrt{\ell}$ contiguous blocks $X_1, X_2, \cdots$ of size $\sqrt{\ell}$ each. Similarly, L_B is partitioned into $\sqrt{\ell}$ contiguous blocks $Y_1, Y_2, \cdots$ of size $\sqrt{\ell}$ each. In parallel for each pair v, w such that v is an

endpoint of an X_i and w is an endpoint of a Y_j, we compute, in $O(\log\log h)$ time and $O(h)$ work, the point $\theta(v,w)$. Thus, if we let Z_{ij} denote the interval on the boundary common to A and B that is defined by the set $\theta(v,w)$ such that $v \in X_i$ and $w \in Y_j$, then after this stage of the computation we know the beginning and end of each such interval Z_{ij}.

The second stage of the computation "fills in the blanks" by doing, in parallel, ℓ recursive calls, one for each X_i, Y_j pair. The call for pair X_i, Y_j returns $\theta(v,w)$ for all $v \in X_i$ and $w \in Y_j$ (of course each such $\theta(v,w)$ is in Z_{ij}).

The time and work complexities of the above method satisfy the recurrences:

$$T(\ell,h) \le T(\sqrt{\ell},h) + c_1 \log\log h,$$

$$W(\ell,h) \le \max\left\{ c_2 \ell h, \sum_{i,j} W(\sqrt{\ell}, |Z_{ij}|) \right\},$$

where c_1 and c_2 are constants. The time recurrence implies that $T(\ell,h) = O(\log\log h \log\log \ell)$. That the processor recurrence implies $W(\ell,h) = O(\ell h \log \ell)$ becomes apparent once we observe that $\sum_{i,j} |Z_{ij}| \le 2h\sqrt{\ell}$. The proof of this last fact is similar to that of Lemma 6: $\sum_{i,j} |Z_{ij}|$ is rewritten as $\sum_{i,\delta} |Z_{i,i+\delta}| \le \sum_\delta h \le 2h\sqrt{\ell}$. This completes the proof of the preliminary CRCW-PRAM bound.

To decrease the work done from $O(m^2 \log m)$ down to $O(m^2 \log\log m)$ (which would imply the bound we claimed in the abstract of this paper), we use a partitioning scheme similar to the ones we used in the CREW-PRAM method, in §6.2. We partition the common boundary into $\log m$ contiguous blocks $J_1, \cdots, J_{\log m}$ of size $m/\log m$ each, then we create $\log m$ subproblems where the ith one consists of computing $\theta_{J_i}(v,w)$ for all $v \in S(L_A, \log m)$ and $w \in S(L_B, \log m)$. We solve in parallel all such subproblems using the preliminary scheme of the previous paragraph, then we "collect answers": for each $v \in S(L_A, \log m)$ and $w \in S(L_B, \log m)$ we compute the correct $\theta(v,w)$ from among $\theta_{J_1}(v,w), \cdots, \theta_{J_{\log m}}(v,w)$. As in §6.2, the $\theta(v,w)$'s so computed define a partition of the common boundary into Z_{ij}'s, whose corresponding subproblems we solve as in the schemes of §6.2: if a Z_{ij} is "small" (i.e., $\le \log m$) then we solve it using the preliminary algorithm; otherwise we partition it into small pieces, solve each of them using the preliminary algorithm; and then collect answers. An analysis like those of §6.2 reveals that the work done is $O(m^2 \log\log m)$, while the time complexity remains $O((\log\log m)^2)$.

Of course the same algorithm as above yields different complexity bounds when we use in it other CRCW-PRAM methods for computing the min of h objects.

8. Conclusion. We gave a number of PRAM algorithms for the string editing problem. The algorithms were fast and efficient, but the best *time* × *processors* bound still did not match the time complexity of the best serial algorithm for the problem [14], [23].

Acknowledgments. The authors are grateful to the referees and to Danny Chen for their careful reading and useful comments, and to Alok Aggarwal for pointing out an error in an earlier analysis of §7.

A referee pointed out that ideas similar to those in §2 were independently found by Baruch Schieber and Uzi Vishkin.

REFERENCES

[1] A.V. AHO, D.S. HIRSCHBERG, AND J.D. ULLMAN, *Bounds on the complexity of the longest common subsequence problem*, J. Assoc. Comput. Mach., 23 (1976), pp. 1-12.

[2] A. AGGARWAL, M. KLAWE, S. MORAN, P. SHOR, AND R. WILBER, *Geometric applications of a matrix searching algorithm*, Algorithmica, 2 (1987) pp. 209-233.

[3] A. AGGARWAL AND J. PARK, *Notes on searching in multidimensional monotone arrays*, in Proc. 29th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1988, pp. 497–512.

[4] ———, *Parallel searching in multidimensional monotone arrays*, unpublished manuscript, May 31, 1989.

[5] A.V. AHO, J.E. HOPCROFT, AND J.D. ULLMAN, *The Design and Analysis of Computer Algorithms*, Addison-Wesley, Reading, MA, 1974.

[6] M.J. ATALLAH, R. COLE, AND M.T. GOODRICH, *Cascading divide-and-conquer: A technique for designing parallel algorithms*, in Proc. 28th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1987, pp. 151–160.

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

[8] R. COLE, *Parallel merge sort*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986, pp. 511–516.

[9] H. FUCHS, Z.M. KEDEM, AND S.P. USELTON, *Optimal surface reconstruction from planar contours*, Comm. Assoc. Comput. Mach., 20 (1977), pp. 693-702.

[10] Z.M. KEDEM AND H. FUCHS, *On finding several shortest paths in certain graphs*, in Proc. 18th Allerton Conference on Communication, Control, and Computing, October 1980, pp. 677–683.

[11] R.E. LADNER AND M.J. FISCHER, *Parallel prefix computation*, J. Assoc. Comput. Mach., 27 (1980), pp. 831–838.

[12] G. LANDAU AND U. VISHKIN, *Introducing efficient parallelism into approximate string matching and a new serial algorithm*, in Proc. 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 220-230.

[13] H. M. MARTINEZ, ED., *Mathematical and computational problems in the analysis of molecular sequences*, Bull. Math. Biol. (Special Issue Honoring M. O. Dayhoff), 46 (1984).

[14] W. J. MASEK AND M. S. PATERSON, *A faster algorithm computing string edit distances*, J. Comput. System Sci., 20 (1980), pp. 18–31.

[15] T. R. MATHIES, *A fast parallel algorithm to determine edit distance*, Tech. Report CMU-CS-88-130, Department of Computer Science, Carnegie Mellon University, Pittsburgh, PA, April 1988.

[16] S. B. NEEDLEMAN AND C.D. WUNSCH, *A general method applicable to the search for similarities in the amino-acid sequence of two proteins*, J. Molecular Biol., 48 (1973), pp. 443–453.

[17] S. RANKA AND S. SAHNI, *String editing on an* SIMD *hypercube multicomputer*, Tech. Report 88-29, Department of Computer Science, University of Minnesota, Minneapolis, MN, March 1988; J. Parallel Distributed Comput., submitted.

[18] D. SANKOFF, *Matching sequences under deletion-insertion constraints*, Proc. Nat. Acad. Sci. U.S.A., 69 (1972), pp. 4–6.

[19] D. SANKOFF AND J. B. KRUSKAL, EDS., *Time Warps, String Edits and Macromolecules: The Theory and Practice of Sequence Comparison*, Addison-Wesley, Reading, MA, 1983.

[20] P.H. SELLERS, *An algorithm for the distance between two finite sequences*, J. Combin. Theory, 16 (1974), pp. 253–258.

[21] ———, *The theory and computation of evolutionary distance: Pattern recognition*, J. Algorithms, 1 (1980), pp. 359–373.

[22] E. UKKONEN, *Finding approximate patterns in strings*, J. Algorithms, 6 (1985), pp. 132–137.

[23] R. A. WAGNER AND M. J. FISCHER, *The string to string correction problem*, J. Assoc. Comput. Mach., 21 (1974), pp. 168–173.

[24] C.K. WONG AND A.K. CHANDRA, *Bounds for the string editing problem*, J. Assoc. Comput. Mach., 23 (1976), pp. 13–16.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 989–999, December 1990

AN IMPROVED ALGORITHM FOR APPROXIMATE STRING MATCHING*

ZVI GALIL† AND KUNSOO PARK‡

Abstract. Given a text string, a pattern string, and an integer k, a new algorithm for finding all occurrences of the pattern string in the text string with at most k differences is presented. Both its theoretical and practical variants improve upon the known algorithms.

Key words. approximate string matching, edit distance, edit sequence

AMS(MOS) subject classifications. 68Q20, 68Q25, 68U15

1. Introduction.

The *edit distance* between a text string $x = x_1x_2 \cdots x_n$ and a pattern string $y = y_1y_2 \cdots y_m$ over an alphabet is the minimum number of *differences* between them such that a correction of the differences converts one string into the other. A *difference* is one of the following:

(1) A character of the pattern corresponds to a different character of the text.

(2) A character of the text corresponds to no character in the pattern.

(3) A character of the pattern corresponds to no character in the text.

Example 1. Let $x =$ `abcdefg`, and $y =$ `ahcefig`. The edit distance between x and y is three, as shown in Fig. 1. The matched characters are indicated by vertical lines. The character `b` in x corresponds to `h` in y (a difference of type (1)), `d` in x to no character in y (a difference of type (2)), and `i` in y to no character in x (a difference of type (3)).

```
x:  a  b  c  d  e  f     g
    |     |     |  |     |
y:  a  h  c     e  f  i  g
```

FIG. 1. *The edit distance between x and y.*

An *edit operation* is an operation which corrects a difference. *Change*, *deletion*, and *insertion* are the edit operations corresponding to the three types of differences. The *edit sequence* between the text and the pattern is the sequence of edit operations for converting one string into the other which realizes the edit distance. Algorithms for finding the edit distance and the edit sequence were given in [9] and [8].

In this paper we are interested in a more general problem; that is, to find all occurrences of the pattern in the text with at most k differences ($k \leq m \leq n$), which is called the *string matching with k differences*. Closely related is the *string matching with k mismatches*, in which only the difference of type (1) is allowed. Together these two problems are called *approximate string matching*.

---

*Received by the editors April 4, 1989; accepted for publication (in revised form) February 13, 1990. This work was supported in part by National Science Foundation grants CCR-86-05353 and CCR-88-14977.

†Department of Computer Science, Columbia University, New York, New York 10027 and Department of Computer Science, Tel-Aviv University, Tel-Aviv, Israel.

‡Department of Computer Science, Columbia University, New York, New York 10027.

For the problem of string matching with k differences Landau and Vishkin provided two algorithms [4] and [5]. Their first algorithm consists of text processing of time bound $O(k^2n)$ and preprocessing of the pattern which has a practical variant and a theoretical variant, depending on the use of a suffix tree and the lowest common ancestor algorithm. Their second algorithm consists of text processing of time bound $O(kn)$ and preprocessing of both text and pattern by using a suffix tree and the lowest common ancestor algorithm, which made it less suitable for practical use. The two algorithms are incomparable for general alphabets. We present a new algorithm whose practical and theoretical variants improve upon both [4] and [5]. Our algorithm consists of text processing of time bound $O(kn)$ and preprocessing of the pattern which has practical and theoretical variants, as does the first algorithm of Landau and Vishkin. The time bounds of the algorithms are summarized in Table 1, where $\widetilde{m}$ is the minimum of m and the size of the alphabet. See [2] for a survey of approximate string matching.

TABLE 1
The time bounds of the algorithms.

Algorithm	Practical	Theoretical
[4]	$O(k^2n + m^2)$	$O(k^2n + m \log \widetilde{m})$
[5]	—	$O(kn + n \log \widetilde{m})$
New	$O(kn + m^2)$	$O(kn + m \log \widetilde{m})$

2. $O(mn)$ algorithms. The ith character of a string x is denoted by x_i. A substring of x from the ith through the jth characters is denoted by $x_i \cdots x_j$. If the minimum number of differences between the pattern y and any substring of the text x ending at x_j is less than k, we say that y *occurs at position j of x with at most k differences*. The problem of string matching with k differences is defined as follows: given a text x of length n, a pattern y of length m, and an integer k ($k \leq m \leq n$), find all positions of x where y occurs with at most k differences. Variations of [9] and [8] solve the string matching with k differences in time $O(mn)$ as follows. (See also [4], [5].)

Let $D(i, j)$, $0 \leq i \leq m$ and $0 \leq j \leq n$, be the minimum number of differences between $y_1 \cdots y_i$ and any substring of x ending at x_j. For $1 \leq i \leq m$ and $1 \leq j \leq n$, the differences between $y_1 \cdots y_i$ and $x_h \cdots x_j$ for some h, $1 \leq h \leq j$ are either

(i) differences between $y_1 \cdots y_{i-1}$ and $x_h \cdots x_{j-1}$ + the difference between y_i and x_j, or

(ii) differences between $y_1 \cdots y_i$ and $x_h \cdots x_{j-1}$ + a difference of type (2) at x_j, or

(iii) differences between $y_1 \cdots y_{i-1}$ and $x_h \cdots x_j$ + a difference of type (3) at y_i.

Thus, $D(i, j)$ is determined by the three entries $D(i-1, j)$, $D(i, j-1)$, and $D(i-1, j-1)$:

$$D(i, j) = \min(D(i-1, j-1) + \delta_{ij}, D(i, j-1) + 1, D(i-1, j) + 1),$$

where $\delta_{ij} = 0$ if $x_j = y_i$; it is 1 otherwise. $D(i, 0) = i$ for $0 \leq i \leq m$ because $y_1 \cdots y_i$ differs from the empty text by i differences of type (3). $D(0, j) = 0$ for $0 \leq j \leq n$ because the empty pattern occurs at any position of the text. $D(m, j) \leq k$ if and only if the pattern occurs at position j of the text with at most k differences. Figure 2

Algorithm MN1

for $i \leftarrow 0$ **to** m **do** $D(i,0) \leftarrow i$ **end for**
for $j \leftarrow 0$ **to** n **do** $D(0,j) \leftarrow 0$ **end for**
for $j \leftarrow 1$ **to** n **do**
  for $i \leftarrow 1$ **to** m **do**
    $row \leftarrow D(i-1,j)+1$
    $col \leftarrow D(i,j-1)+1$
    if $x_j = y_i$ **then** $diag \leftarrow D(i-1,j-1)$
    else $diag \leftarrow D(i-1,j-1)+1$ **end if**
    $D(i,j) \leftarrow \min(row, col, diag)$
  end for
end for

FIG. 2. *The algorithm* MN1.

shows the dynamic programming algorithm MN1, which is a variation of Wagner and Fischer's algorithm [9]. It computes the D table column by column. Since there are $O(mn)$ entries and each entry takes constant time to be computed, algorithm MN1 takes time $O(mn)$.

Example 2. Let $x =$ abbdadcbc, $y =$ adbbc, and $k = 2$. Table 2 shows table $D(i,j)$, $0 \leq i \leq 5$ and $0 \leq j \leq 9$. The pattern occurs at positions 3, 4, 7, 8, and 9 of the text with at most two differences.

TABLE 2
Table $D(i,j)$.

		0	1	2	3	4	5	6	7	8	9
	D		a	b	b	d	a	d	c	b	c
0		0	0	0	0	0	0	0	0	0	0
1	a	1	0	1	1	1	0	1	1	1	1
2	d	2	1	1	2	1	1	0	1	2	2
3	b	3	2	1	1	2	2	1	1	1	2
4	b	4	3	2	1	2	3	2	2	1	2
5	c	5	4	3	2	2	3	3	2	2	1

LEMMA 1 [8]. *For every* $D(i,j)$, $1 \leq i \leq m$ *and* $1 \leq j \leq n$, $D(i,j) = D(i-1, j-1)$ *or* $D(i,j) = D(i-1,j-1)+1$.

Let *D-diagonal* d be the entries of table $D(i,j)$ such that $j - i = d$. Lemma 1 suggests a more compact way to store the information of the D table. For each D-diagonal we store only the positions where the value increases. For a D-diagonal d and a difference e, let $C(e,d)$ be the largest column j such that $D(j-d,j) = e$. In other words, the entries of value e on D-diagonal d end at column $C(e,d)$. Note that $C(e,d)-d$ is the row of the last entry on D-diagonal d whose value is e. Let *C-diagonal* c be the entries of table $C(e,d)$ such that $e + d = c$. The definition of $C(e,d)$ implies that the minimum number of differences between $y_1 \cdots y_{C(e,d)-d}$ and any substring of the text ending at $x_{C(e,d)}$ is e, and $x_{C(e,d)+1} \neq y_{C(e,d)+1-d}$. $C(e,d) = m + d$ for some $e \leq k$ if and only if the pattern occurs at position $m + d$ of the text with at most k differences.

Example 3. Consider $x =$ abbdadcbc, $y =$ adbbc, and $k = 2$ again. Table 3 shows the C table, where columns are D-diagonals and rows are differences. The

initial entries (-1, $-\infty$, and the first row) will be explained below. Note that on D-diagonal 4 in Table 2 there is no entry of value 2, in which case we set $C(2,4) = 9$, the last column of D-diagonal 4. For D-diagonal $d = -2, -1, 2, 3$, and 4, $C(2,d) = 5+d$. Thus the pattern occurs at positions 3, 4, 7, 8, and 9 of the text with at most two differences.

TABLE 3
Table $C(e,d)$.

	C	d										
		−3	−2	−1	0	1	2	3	4	5	6	7
	−1			$-\infty$	−1	0	1	2	3	4	5	6
e	0		$-\infty$	−1	1	1	2	3	6	5	6	
	1	$-\infty$	−1	3	3	2	4	6	9	8		
	2	−1	3	4	4	4	7	8	9			

There are three types of D-diagonals with respect to the C table:

(i) For $-k \le d \le -1$, we compute $d+k+1$ entries $C(e,d)$, $|d| \le e \le k$, because D-diagonal d starts with value $|d|$.

(ii) For $0 \le d \le n-m$, we compute $k+1$ entries $C(e,d)$, $0 \le e \le k$.

(iii) For $n-m+1 \le d \le n-m+k$, we compute $(n-m+k)-d+1$ entries $C(e,d)$, $0 \le e \le (n-m+k)-d$ because D-diagonal $n-m$ is the last D-diagonal for which we want to compute C, and D-diagonal d may affect the values of D-diagonal $n-m$ by differences of type (3).

Thus the shape of the C table is a parallelogram. In order to compute the C table directly without using the D table, we have the following initial entries. For D-diagonal $d \ge 0$, the initial value of the D-diagonal is 0 at column d (i.e., $D(0,d) = 0$), so we assign $d-1$ to $C(-1,d)$, which indicates that imaginary entries of value -1 end at column $d-1$. Since the initial value of D-diagonal d, $-(k+1) \le d \le -1$, is $|d|$ at column 0, we assign -1 to $C(|d|-1,d)$. We also assign $-\infty$ to $C(|d|-2,d)$, $-(k+1) \le d \le -1$, so that they are properly initialized.

The C table is computed by the algorithm MN2 (Fig. 3), which is a variation of Ukkonen's algorithm [8]. It proceeds C-diagonal by C-diagonal. Now we show that algorithm MN2 computes the C table correctly. Consider the computation of $C(e,c-e)$ for $0 \le c \le n-m+k$ and $0 \le e \le k$. For notational convenience let $d = c-e$. Recall that $C(e,d)$ is the column of the last entry on D-diagonal d whose value is e. Assume by induction that $C(e-1,d-1)$, $C(e-1,d)$, and $C(e-1,d+1)$ were computed correctly. This means that in the D table the entries of value $e-1$ reach column $C(e-1,d-1)$ on D-diagonal $d-1$, column $C(e-1,d)$ on D-diagonal d, and column $C(e-1,d+1)$ on D-diagonal $d+1$. Let col' be the maximum of $C(e-1,d-1)+1$, $C(e-1,d)+1$, and $C(e-1,d+1)$. Note the col' may be greater than either n or $m+d$ (the last position of either string), in which case we set $C(e,d) = \min(m+d,n)$, the last column of D-diagonal d in the D table. If col' is less than or equal to both n and $m+d$, $D(col'-d,col')$ gets value e from one of the last entries of value $e-1$ on D-diagonals $d-1$, d, and $d+1$ by one of the three types of differences. The entries of value e on D-diagonal d continue to column col'' such that $x_{col''+1} \ne y_{col''+1-d}$ is the first mismatch for $col'' \ge col'$, or col'' is either n or $m+d$. Then $C(e,d) = col''$.

Example 4. Consider the computation of $C(2,2)$ in Example 3. $C(1,1)+1 = 3$, $C(1,2)+1 = 5$, and $C(1,3) = 6$. So col' for $C(2,2)$ is 6, and $D(4,6)$ gets value 2 from

Algorithm MN2

// initialization //
for $d \leftarrow 0$ **to** $n - m + k + 1$ **do** $C(-1, d) \leftarrow d - 1$ **end for**
for $d \leftarrow -(k+1)$ **to** -1 **do**
  $C(|d| - 1, d) \leftarrow -1$
  $C(|d| - 2, d) \leftarrow -\infty$
end for
for $c \leftarrow 0$ **to** $n - m + k$ **do**
  for $e \leftarrow 0$ **to** k **do**
    $d \leftarrow c - e$
    $col \leftarrow \max(C(e-1, d-1) + 1, C(e-1, d) + 1, C(e-1, d+1))$
    while $col < n$ **and** $col - d < m$ **and** $x_{col+1} = y_{col+1-d}$ **do**
      $col \leftarrow col + 1$
    end while
    $C(e, d) \leftarrow \min(col, m + d, n)$
  end for
end for

FIG. 3. *The algorithm* MN2.

$D(3, 6) = 1$. There is a match $x_7 = y_5$ at column 7 of D-diagonal 2, which is the last match on the D-diagonal. Thus $C(2, 2) = 7$.

LEMMA 2. *The characters of the text which are compared with the pattern in the computation of C-diagonal c are at most $x_{c+1}, \cdots, x_{c+m}$ ($x_{c+1}, \cdots, x_n$ if $c + m > n$).*

Proof. We show that in order to compute $C(e, c - e)$ for $0 \leq e \leq k$, at most $x_{c+1}, \cdots, x_{m+c-e}$ are compared with the pattern. To compute $C(0, c)$, we start with the comparison of y_1 and x_{c+1}, and we may continue up to the comparison of y_m and x_{c+m}, which is the last one on D-diagonal c. Thus we compare at most $x_{c+1}, \cdots, x_{c+m}$ with the pattern for $C(0, c)$. To compute $C(e, c - e)$ for $0 < e \leq k$, the first position of the text to be compared is greater than or equal to $c + 1$: the computation of $C(0, c - e)$ starts at text position $c - e + 1$, and there is at least one entry of each value e', $0 \leq e' < e$, on D-diagonal $c - e$. The entries of value e on D-diagonal $c - e$ may continue to $D(m, m + c - e)$, the last entry on D-diagonal $c - e$. Thus, at most $x_{c+1}, \cdots, x_{m+c-e}$ are compared for $C(e, c - e)$. If any position of the text is greater than n, the last position to be considered should be n. □

LEMMA 3. *During the computation of C-diagonal c,*

(1) *The positions of the characters of the text which are actually compared with the pattern are nondecreasing.*
(2) *The repetitions of text positions occur at most k times.*

Proof. Let j be the text position where a mismatch occurred in the computation of $C(e, c - e)$ for $0 \leq e < k$ (i.e., $C(e, c - e) = j - 1$). We show that the first position of the text to be considered for $C(e + 1, c - e - 1)$ is at least j. At the beginning of the computation of $C(e + 1, c - e - 1)$, $col \geq C(e, c - e)$. The first position of the text $col + 1$ satisfies the following:

$$col + 1 \geq C(e, c - e) + 1 = j.$$

Since the repetition of a text position occurs only at the first comparison for $C(e, c-e)$, $1 \leq e \leq k$, there are at most k repetitions. □

By Lemmas 2 and 3 the computation of each C-diagonal takes time $O(m)$. Since there are $n - m + k + 1$ C-diagonals, algorithm MN2 takes time $O(mn)$.

3. The new algorithm. The algorithm consists of preprocessing of the pattern followed by processing of the text. In the preprocessing we build an upper triangular table *Prefix*(i,j), $1 \leq i < j \leq m$, where *Prefix*(i,j) is the length of the longest common prefix of $y_i \cdots y_m$ and $y_j \cdots y_m$. This table is used for the comparison of two substrings of the pattern during the text processing. The details of the preprocessing will be discussed in the next section.

The text processing is based on the second algorithm in the previous section. It consists of $n-m+k+1$ iterations, one for each C-diagonal, as algorithm MN2 does. Whereas algorithm MN2 relies only on direct comparisons of the text with the pattern, the new algorithm uses both direct comparisons and lookups of the *Prefix* table. If a substring of the text had matches with a substring of the pattern, the algorithm looks up the *Prefix* table for the substring of the text. Otherwise, it directly compares the text with the pattern. For the matched part of the text the algorithm compares two substrings of the pattern instead of comparing a substring of the pattern with a substring of the text. This technique, which first appeared in the Knuth–Morris–Pratt algorithm [3], was also used in [4] and [7].

A *reference triple* (u,v,w) consists of a start position u, an end position v, and a D-diagonal w such that substring $x_u \cdots x_v$ of the text matches substring $y_{u-w} \cdots y_{v-w}$ of the pattern and $x_{v+1} \neq y_{v+1-w}$. Note that w is the D-diagonal where the match occurred. We call $y_{u-w} \cdots y_{v-w}$ the *reference* of $x_u \cdots x_v$. If $u > v$ in a triple (u,v,w), the triple is called *null*, and it indicates that $[u,v]$ is an empty interval and $x_{v+1} \neq y_{v+1-w}$. The idea of triples that are equivalent to reference triples appeared in [4].

At iteration c we compute C-diagonal c that is $C(e,c-e)$, $0 \leq e \leq k$. Let q be the text position such that x_{q+1} is the rightmost character of the text which was compared with the pattern before iteration c (i.e., x_{q+1} had a mismatch). Suppose that from previous iterations we have $k+1$ reference triples (u_0,v_0,w_0), (u_1,v_1,w_1), $\cdots$, (u_k,v_k,w_k) such that the set of intervals $[u_0,v_0]$, $[u_1,v_1]$, $\cdots$, $[u_k,v_k]$ is a partition of interval $[c,q]$ with a possible hole between v_e and u_{e+1} for $0 \leq e < k$ (i.e., either $u_{e+1} = v_e + 1$ or $u_{e+1} = v_e + 2$). Initially, $q = 0$ and all triples are $(0,0,0)$.

Let t be the current text position ($col + 1$ in Figs. 3 and 4) in the computation of $C(e,c-e)$ for $0 \leq e \leq k$. Again, let $d = c - e$ for convenience. To compute $C(e,d)$, we look for the first mismatch $x_j \neq y_{j-d}$ for $j \geq t$. Then $C(e,d)$ will be $j-1$. If $t > q$, we have no reference triples for x_t. So we compare the text with the pattern until there is a mismatch. While $t \leq q$, we compare the pattern with references unless t is the position of a hole, in which case x_t is directly compared with the pattern. If t is within the interval of a reference triple (u_r,v_r,w_r) for some $0 \leq r \leq k$, we look up the *Prefix* table. The current pattern position p is $t-d$, and the reference position corresponding to t is $t-w_r$. We look at *Prefix*$(p,t-w_r)$. Let f be $v_r - t + 1$, the length of the reference from $t-w_r$ to $v_r - w_r$. Let g be *Prefix*$(p,t-w_r)$, the length of the longest common prefix of $y_p \cdots y_m$ and $y_{t-w_r} \cdots y_m$. There are three cases:

(i) $f < g$: text $x_t \cdots x_{t+f-1}$ matches pattern $y_p \cdots y_{p+f-1}$, but $x_{t+f} \neq y_{p+f}$ because $x_{t+f} \neq y_{t+f-w_r}$ by the definition of reference triples, and $y_{t+f-w_r} = y_{p+f}$ since $f < g$.

(ii) $f = g$: text $x_t \cdots x_{t+f-1}$ matches pattern $y_p \cdots y_{p+f-1}$, and x_{t+f} may or may not match y_{p+f} because $x_{t+f} \neq y_{t+f-w_r}$ and $y_{t+f-w_r} \neq y_{p+f}$.

(iii) $f > g$: text $x_t \cdots x_{t+g-1}$ matches pattern $y_p \cdots y_{p+g-1}$, but $x_{t+g} \neq y_{p+g}$ because $x_{t+g} = y_{t+g-w_r}$ and $y_{t+g-w_r} \neq y_{p+g}$.

In cases (i) and (iii) we have found j that is $t + \min(f,g)$. In case (ii) we continue at position $t+f$.

After iteration c we update reference triples for the next iteration. Let s_e,

```
Algorithm KN
initializations
for c ← 0 to n − m + k do
  r ← 0
  for e ← 0 to k do
    d ← c − e
    col ← max(C(e − 1, d − 1) + 1, C(e − 1, d) + 1, C(e − 1, d + 1))
    s_e ← col + 1
    found ← false
    while not found do
      if within(col + 1, k, r) then
        f ← v_r − col
        g ← Prefix(col + 1 − d, col + 1 − w_r)
        if f = g then
          col ← col + f
        else
          col ← col + min(f, g)
          found ← true
        end if
      else
        if col < n and col − d < m and x_{col+1} = y_{col+1−d} then
          col ← col + 1
        else
          found ← true
        end if
      end if
    end while
    C(e, d) ← min(col, m + d, n)
    // update reference triple (u_e, v_e, w_e) //
    if v_e ≥ C(e, d) then
      if e = 0 then u_e ← c + 1
      else u_e ← max(u_e, v_{e−1} + 1) end if
    else
      v_e ← C(e, d)
      w_e ← d
      if e = 0 then u_e ← c + 1
      else u_e ← max(s_e, v_{e−1} + 1) end if
    end if
  end for
end for
```

FIG. 4. *The algorithm* KN.

$0 \leq e \leq k$, be the first position of the text which was considered for $C(e,d)$. $C(e,d)$ itself is the last position where a series of matches (possibly empty) ended. Namely, $x_{s_e}, \cdots, x_{C(e,d)}$ had matches with the pattern if $s_e \leq C(e,d)$. Therefore, triples $(s_e, C(e,d), d)$, $0 \leq e \leq k$, are reference triples which came up from the computation of C-diagonal c. However, triples $(s_e, C(e,d), d)$, $0 \leq e \leq k$, cannot immediately become new reference triples for the next iteration, since there may be multiple holes between $C(e,d)$ and s_{e+1} (i.e., $s_{e+1} > C(e,d)+2$). We combine the old reference triple

(u_e, v_e, w_e) and the reference triple $(s_e, C(e,d), d)$ to obtain the new reference triple (u'_e, v'_e, w'_e) for each $0 \le e \le k$. Two triples (u_e, v_e, w_e) and $(s_e, C(e,d), d)$ compete for (u'_e, v'_e, w'_e), and the one with larger end position (i.e., v_e vs. $C(e,d)$) wins. When $v_e = C(e,d)$, the tie can be broken arbitrarily (in Fig. 4 we choose (u_e, v_e, w_e) as the winner to minimize update operations). New v'_e and w'_e are those of the winner. It follows by induction on the iteration number c that after the update, v'_e is the maximum of $C(e, i-e)$, $0 \le i \le c$. New u'_e is $c+1$ if $e = 0$. If $e > 0$, there are two cases for u'_e. In each case we show that (u'_e, v'_e, w'_e) is a reference triple and u'_e is either $v'_{e-1}+1$ or $v'_{e-1}+2$.

(i) If (u_e, v_e, w_e) is the winner, u'_e is the maximum of u_e and $v'_{e-1}+1$.
  (a) (u'_e, v'_e, w'_e) is a reference triple because (u_e, v_e, w_e) is a reference triple and $u'_e \ge u_e$.
  (b) u_e is either $v_{e-1}+1$ or $v_{e-1}+2$ from previous iterations, and $v_{e-1} \le v'_{e-1}$ by the previous competition. Thus, $u_e > v'_{e-1}+1$ only when $u_e = v_{e-1}+2$ and $v_{e-1} = v'_{e-1}$, in which case $u_e = v'_{e-1}+2$.

(ii) If $(s_e, C(e,d), d)$ is the winner, u'_e is the maximum of s_e and $v'_{e-1}+1$.
  (a) (u'_e, v'_e, w'_e) is a reference triple because $(s_e, C(e,d), d)$ is a reference triple and $u'_e \ge s_e$.
  (b) s_e is the maximum of $C(e-1, d+1)+1$, $C(e-1,d)+2$, and $C(e-1, d-1)+2$. v'_{e-1} is the maximum of $C(e-1,d+1)$, $C(e-1,d)$, $C(e-1,d-1)$, $\cdots$, $C(e-1,-(e-1))$. Thus, $s_e > v'_{e-1}+1$ when $s_e = C(e-1,d)+2$ and $v'_{e-1} = C(e-1,d)$, or when $s_e = C(e-1,d-1)+2$ and $v'_{e-1} = C(e-1,d-1)$. In both cases, $s_e = v'_{e-1}+2$.

Therefore, (u'_0, v'_0, w'_0), (u'_1, v'_1, w'_1), $\cdots$, (u'_k, v'_k, w'_k) are reference triples, and the set of intervals $[u'_0, v'_0]$, $[u'_1, v'_1]$, $\cdots$, $[u'_k, v'_k]$ is a partition of interval $[c+1, q']$ with a possible hole between v'_e and u'_{e+1} for $0 \le e < k$, where q' is the largest end position of the triples (q' may not be v'_k if triple (u'_k, v'_k, w'_k) is null). Instead of updating all reference triples at the end of iteration c, we can update the reference triple (u_e, v_e, w_e) after the computation of $C(e,d)$. If the old reference triple (u_e, v_e, w_e) is the winner, triple $(s_e, C(e,d), d)$ is simply discarded. Otherwise, the text position to be considered next is greater than $C(e,d)$ by Lemma 3.1, so we can update (u_e, v_e, w_e) safely.

Example 5. Consider $x =$ `abbdadcbc`, $y =$ `adbbc`, and $k = 2$. Table 4 shows the reference triples at the beginning of iteration c, $1 \le c \le 6$. The triples marked with * are null triples.

TABLE 4
Reference triples.

c	Reference triples 0	1	2
1	$(1,1,0)$	$(2,3,-1)$	$(4,3,-2)^*$
2	$(2,1,0)^*$	$(2,3,-1)$	$(5,4,-1)^*$
3	$(3,2,2)^*$	$(3,3,-1)$	$(5,4,-1)^*$
4	$(4,3,3)^*$	$(4,4,2)$	$(5,4,-1)^*$
5	$(5,6,4)$	$(7,6,3)^*$	$(7,7,2)$
6	$(6,6,4)$	$(8,9,4)$	$(10,8,3)^*$

The algorithm KN in Fig. 4 shows the text processing. Procedure *within*(t,k,r) tests if text position t is within an interval of reference triples, in which case r is the

```
procedure within(t, k, r)
  while r ≤ k and t > v_r do
    r ← r + 1
  end while
  if r > k then return(false)
  else
    if t ≥ u_r then return(true)
    else return(false) end if
  end if
end
```

FIG. 5. *The procedure "within(t, k, r)."*

index of the reference triple within whose interval text position t is. At the beginning of iteration c, r is 0. By Lemma 3.1, r never decreases during iteration c. If $t > v_i$ for all $0 \leq i \leq k$, obviously $t > q$; $within(t, k, r)$ returns **false**. If $t \leq q$, it increases r by 1 until $t \leq v_r$. If $u_r \leq t$, t is in interval $[u_r, v_r]$; it returns **true**. If $t < u_r$, t is the position of a hole; it returns **false**. Figure 5 shows the procedure $within(t, k, r)$. The text position q is implicitly maintained by the reference triples.

At iteration c the number of repetitions of the **while** loop in algorithm KN is the number of direct comparisons plus the number of lookups of the *Prefix* table. Direct comparisons are counted in two ways:

(i) If $t > q + 1$ (i.e., x_t is a new character), the comparison is charged to text position t.

(ii) If $t \leq q + 1$ (i.e., x_t was compared before iteration c), the comparison is charged to C-diagonal c.

When $t > q + 1$, there can be at most k repetitions of text position t during iteration c by Lemma 3.2. At the next iteration the text position belongs to (ii). Thus there are $O(kn)$ comparisons for the whole text processing by rule (i). In interval $[c, q + 1]$ there are at most k holes from the reference triples and another hole at $q+1$. A direct comparison at a hole either increases e (when a mismatch occurs) or causes passing the hole (when a match occurs). Hence, $O(k)$ comparisons are charged to C-diagonal c by rule (ii). Table lookups are charged to C-diagonal c. A lookup of the table increases either e (when $f \neq g$) or r (when $f = g$); $O(k)$ lookups are charged to C-diagonal c. At iteration c, procedure *within* also takes time $O(k)$ because r increases from 0 to at most $k + 1$. Since there are $n - m + k + 1$ C-diagonals, the total time of the text processing is $O(kn)$.

The text processing maintains the C table and $k + 1$ reference triples. To find edit distances (i.e., string matching with k differences) we keep only two previous C-diagonals. Thus, the space required for the text processing is $O(k)$. If we want to find both edit distances and edit sequences, we need to keep k C-diagonals [8], which leads to $O(k^2)$ space.

4. The preprocessing. In the preprocessing of pattern y we compute upper triangular table $Prefix(i, j)$, $1 \leq i < j \leq m$, where $Prefix(i, j)$ is the length of the longest common prefix of $y_i \cdots y_m$ and $y_j \cdots y_m$. The procedure in Fig. 6 computes $Prefix(i, j)$ diagonal by diagonal. For each diagonal d it starts to compare y_1 with y_{1+d} and proceeds on the diagonal until there is a mismatch $y_c \neq y_{c+d}$. Then $Prefix(1, 1 + d) = c - 1$, $Prefix(2, 2 + d) = c - 2$, $\cdots$, $Prefix(c, c + d) = 0$. It resumes the comparison with y_{c+1} and y_{c+1+d}, and repeats until it reaches the end of the pattern. If the procedure makes c comparisons in the inner **while** loop, it fills in c entries of

```
for d ← 1 to m − 1 do
  i ← 0
  while i + d < m do
    c ← 1
    while i + c + d ≤ m and y_{i+c} = y_{i+c+d} do
      c ← c + 1
    end while
    for j ← 1 to c do
      Prefix(i + j, i + j + d) ← c − j
    end for
    i ← i + c
  end while
end for
```

FIG. 6. *The preprocessing.*

the *Prefix* table. Since there are $m(m-1)/2$ entries, the preprocessing takes time and space $O(m^2)$. An alternate computation of the *Prefix* table which also takes time and space $O(m^2)$ appears in [4]. Taking into account both preprocessing and text processing, our algorithm takes time $O(kn + m^2)$ and space $O(m^2)$.

Using a suffix tree and the lowest common ancestor algorithm ([2], [4], [5]), the time bound of the preprocessing can be reduced to $O(m \log m)$ for general alphabets or to $O(m)$ for alphabets whose size is fixed, and the space bound is reduced to $O(m)$. Since, however, the constant hidden in the suffix tree and the lowest common ancestor algorithm is quite large, it is mostly of theoretical interest. In this case our algorithm takes time $O(kn + m \log \widetilde{m})$ and space $O(m)$.

5. Conclusion. We have presented a new algorithm for the string matching with k differences which improves upon the known algorithms. It is interesting that the time and space bounds of our algorithm are the same as those of Galil and Giancarlo's algorithm [1] for the string matching with k mismatches. In addition to the bounds, they are similar in that both algorithms use the *Prefix* table and maintain $k+1$ references (by a D-diagonal and mismatched text positions in [1], and by reference triples in ours). They are different in that [1] finds start positions of occurrences of the pattern in the text while our algorithm finds end positions of the occurrences, and [1] builds references from one D-diagonal while our algorithm builds them from at most $k+1$ D-diagonals.

In [6] and [8], an additional type of difference was considered:

(4) Two adjacent characters `ab` of the pattern correspond to the transposed characters `ba` of the text.

Transposition is the edit operation which corrects the difference of type (4). Our algorithm with some modifications can be extended to include the difference of type (4). Since a mismatch $x_i \neq y_{i-d}$ may turn out to be a difference of type (4) $x_i x_{i+1} = y_{i+1-d} y_{i-d}$,

$$col \leftarrow \max(C(e-1, d-1) + 1, C(e-1, d) + 1, C(e-1, d+1))$$

should be replaced by

$$i \leftarrow C(e-1, d) + 1$$
$$\textbf{if } x_i x_{i+1} = y_{i+1-d} y_{i-d} \textbf{ then } i \leftarrow i + 1 \textbf{ end if}$$
$$col \leftarrow \max(i, C(e-1, d-1) + 1, C(e-1, d+1)).$$

Now the first position s_e of the text which is considered for $C(e,d)$ can be $v'_{e-1}+3$ because of transposed characters. Thus there are at most two holes between the intervals of reference triples.

Acknowledgment. We thank the referees for helpful comments.

REFERENCES

[1] Z. GALIL AND R. GIANCARLO, *Improved string matching with k mismatches*, SIGACT News, 17 (1986), pp. 52–54.

[2] ———, *Data structures and algorithms for approximate string matching*, J. Complexity, 4 (1988), pp. 33–72.

[3] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT, *Fast pattern matching in strings*, SIAM J. Comput., 6 (1977), pp. 323–350.

[4] G. M. LANDAU AND U. VISHKIN, *Fast string matching with k differences*, J. Comput. System Sci., 37 (1988), pp. 63–78.

[5] ———, *Fast parallel and serial approximate string matching*, J. Algorithms, 10 (1989), pp. 157–169.

[6] R. LOWRANCE AND R. A. WAGNER, *An extension of the string-to-string correction problem*, J. Assoc. Comput. Mach., 22 (1975), pp. 177–183.

[7] M. G. MAIN AND R. J. LORENTZ, *An $O(n \log n)$ algorithm for finding all repetitions in a string*, J. Algorithms, 5 (1984), pp. 422–432.

[8] E. UKKONEN, *Algorithms for approximate string matching*, Inform. and Control, 64 (1985), pp. 100–118.

[9] R. A. WAGNER AND M. J. FISCHER, *The string-to-string correction problem*, J. Assoc. Comput. Mach., 21 (1974), pp. 168–173.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1000–1010, December 1990

LINEAR PROGRAMMING WITH TWO VARIABLES PER INEQUALITY IN POLY-LOG TIME*

GEORGE S. LUEKER†, NIMROD MEGIDDO‡, AND VIJAYA RAMACHANDRAN§

Abstract. The parallel time complexity of the linear programming problem with at most two variables per inequality is discussed. Let n and m denote the number of variables and the number of inequalities, respectively, in a linear programming problem. It is assumed that all inequalities are weak. Under the concurrent-read-exclusive-write PRAM model, an $O((\log m + \log^2 n)\log^2 n)$-time parallel algorithm for deciding feasibility is described. It requires $mn^{O(\log n)}$ processors in the worst case, though it is not known whether this bound is tight. When the problem is feasible, a solution can be computed within the same complexity. Moreover, linear programming problems with at most two nonzero coefficients in the objective function can be solved in poly-log time on a similar number of processors. Consequently, all these problems can be solved sequentially with only $O((\log m + \log^2 n)^2 \log^2 n)$ space. (These bounds assume that numbers take $O(1)$ space, and arithmetic on them takes $O(1)$ time; the problem can still be solved in poly-log space as a function of the input size even if a Turing machine model with rational input is used instead.) It is also shown that if the underlying graph has bounded tree-width and an underlying tree is given, then the feasibility problem is in the class NC.

Key words. parallel computation, linear programming, poly-log time

AMS(MOS) subject classifications. 68Q25, 90G05

1. Introduction.

Dobkin, Lipton, and Reiss [7] first showed that the general linear programming problem was (log-space) hard for P. Combined with Khachiyan's deep result [14] that the problem is in P, this establishes that the problem is P-complete (that is, log-space complete for P). A popular specialization of the general linear programming problem is the problem of solving linear inequalities with at most two variables per inequality (see [15] and the references therein). It is shown in [15] that a system of m linear inequalities in n variables (but at most two nonzero coefficients per inequality) can be solved in $O(mn^3 \log m)$ arithmetic operations and comparisons over any ordered field. It is not known whether the general problem (even only over the rationals) can be solved in less than $p(m, n)$ operations, for any polynomial p.

Throughout the paper we assume that the space to store numbers and the time for arithmetic operations is $O(1)$. Since each expression we compute can be written as an expression tree of height $O(\log n)$ in the input values, the length of numbers only increases by a polynomial factor during the execution of the algorithm, so this assumption does not alter the statements of our results by more than a polynomial factor for the number of processors, or more than a poly-log factor for the time.

In this paper we are interested in the *parallel* computational complexity of the two variables per inequality problem. We first mention some related results which, we hope, shed some light on the parallel complexity of the problem.

* Received by the editors October 17, 1988; accepted for publication (in revised form) October 6, 1989. An earlier version of this paper appeared in the Proceedings of the 18th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1986, pp. 196–205. This work was done while the authors were at the Mathematical Sciences Research Institute, Berkeley, California.

† Department of Information and Computer Science, University of California, Irvine, California 92717. The research of this author was supported by National Science Foundation grant DCR-8509667.

‡ IBM Research, Almaden Research Center, 650 Harry Road, San Jose, California 95120, and School of Mathematical Sciences, Tel Aviv University, Tel Aviv, Israel.

§ Department of Computer Science, University of Texas, Austin, Texas 78712. The research of this author was supported by National Science Foundation grant ECS-8404866, by Joint Services Electronics Program grant N00014-84-CO149, and by an IBM Faculty Development award.

PROPOSITION 1.1. *The problem of finding the minimum value of a general linear function subject to linear inequalities with at most two variables per inequality is P-complete.*

Proof. The proof follows from the result that the problem of finding the *value* of the maximum flow through a capacitated network is P-complete [11]. More specifically, every maximum flow problem can be reduced [8] to a transportation problem, that is, a problem of the form

$$\text{minimize} \quad \sum_{ij} c_{ij}x_{ij}$$

$$\text{subject to} \quad \sum_{j} x_{ij} \leqq a_i, \quad \sum_{i} x_{ij} \geqq b_j, \quad x_{ij} \geqq 0.$$

The dual of the latter has only two variables per inequality. □

In fact, there exist much simpler but yet P-complete linear programming problems with a general objective function and only two variables per inequality. For example, consider the following recursive formula:

$$x_{k+1} = \max(\alpha_k x_k + \beta_k, \gamma_k x_k + \delta_k),$$

where α_k, β_k, γ_k, and δ_k $(k = 1, \cdots, n)$ are given numbers. Given any value of x_1, we want to determine the resulting value of x_n. Let us call this the PROPAGATION problem. The propagation problem can be solved as the following linear programming problem:

$$\text{minimize} \quad \sum_{i} M^i x_i$$

$$\text{subject to} \quad x_{k+1} \geqq \alpha_k x_k + \beta_k, \quad x_{k+1} \geqq \gamma_k x_k + \delta_k,$$

where M is sufficiently large.

PROPOSITION 1.2. PROPAGATION *is P-complete.*

Proof. The proposition follows from the result by Helmbold and Mayr [12] that the 2-processor list scheduling problem is P-complete. The latter amounts to a special case of PROPAGATION: $x_k = |x_{k-1} - T_k| = \max\{x_{k-1} - T_k, -x_{k-1} + T_k\}$, $(k = 1, \cdots, n)$, where $T_1, \cdots, T_n$ are given integers and $x_0 = 0$. □

Interestingly, the subproblem of PROPAGATION where all the α_k's and γ_k's are nonnegative is in NC; this follows immediately from the parallel composition of monotonic piecewise linear functions to be discussed in § 3.

We discuss three different problems:

(i) Deciding feasibility. Here we only need to determine whether there exists a solution to a given set of linear inequalities with at most two variables per inequality.

(ii) Solving inequalities. Here we require that if the system is feasible then some point in the feasible space be determined.

(iii) Optimization. Here we seek to find a point in the feasible space which maximizes a given linear combination of two of the variables. As in the previous case, here we must distinguish between the problem of computing the optimum value of the function and the problem of computing optimum values of the variables.

Deciding feasibility is the core of our algorithm. This is accomplished by computing the *projections* of the set of feasible solutions onto the individual coordinate axes. Using a parallelization of the sequential algorithm of [17], we will compute for each variable x an interval $[x_{\text{low}}, x_{\text{high}}]$, $-\infty \leqq x_{\text{low}} \leqq x_{\text{high}} \leqq \infty$ (possibly empty), so that for every $x' \in [x_{\text{low}}, x_{\text{high}}]$ there is a solution to the system of inequalities with $x = x'$. This part of the algorithm suffices of course for determining whether a given system of linear inequalities (with at most two variables per inequality) has a solution. Later we

shall discuss the problem of finding a feasible solution given the (nonempty) projections onto the axes. Finally, we show how to optimize a linear function with at most two nonzero coefficients subject to such systems of inequalities.

2. Preliminaries. Two characterizations of feasibility of linear inequalities with at most two variables per inequality were given by Nelson [17] and Shostak [19]. Our algorithm is a parallelization of the algorithm in Nelson [17], but our exposition will also make use of the characterization of Shostak [19] and further results by Aspvall and Shiloach [2] which we now describe.[1] A key idea in these papers is the observation that we can combine inequalities to deduce new inequalities. For example, suppose we are given $x+2y \leqq 3$ and $-y+3z \leqq 4$. Eliminating y between these two inequalities, we obtain $x+6z \leqq 11$.

It is convenient to discuss the problem using graph-theoretic terminology. We form what is called the *constraints graph*, denoted G, by creating a vertex for each variable, and an edge for each inequality whose endpoints are the two variables appearing in the inequality. (Note that this is actually a multigraph, since a pair of variables can be involved in many different inequalities.) We henceforth identify variables with the corresponding vertices. To take care of inequalities involving only a single variable, such as $x \leqq 4$, in [19] a dummy vertex v_0 was also added corresponding to a new variable that can only occur with coefficient zero; an edge representing an inequality involving only one variable, say x, runs between x and v_0. Now let L denote a path in G from x to y, that is, a set of inequalities $\alpha_i x_i + \beta_{i+1} x_{i+1} \leqq \gamma_i$ $(i = 0, 1, \cdots, k-1)$, where $x = x_0$ and $y = x_k$. (Note that since G is a multigraph, a path should be thought of as a sequence of edges rather than as a sequence of vertices, to avoid ambiguity.) If $\alpha_i \beta_i < 0$ for $i = 1, \cdots, k-1$, we say P is *admissible*, and then by successive elimination of $x_1, x_2, \cdots, x_{k-1}$ we can deduce a new inequality $\alpha x + \beta y \leqq \gamma$ that is implied by this path. We will call this the *resultant* of the path. If on the other hand there exists an i $(1 \leqq i \leqq k-1)$ such that $\alpha_i \beta_i > 0$, we say the path is inadmissible, and then we cannot deduce a resultant.

The case in which the endpoints of the path are the same variable deserves special attention; in this case we call the path a *loop*. Then the resultant involves only a single variable, that is, it is a bound of the form $\alpha x \leqq \gamma$. It is possible that the resultant could be an inherent contradiction—this occurs if $\alpha = 0$ and $\gamma < 0$. For instance, the path $-x+2y \leqq -7$ and $-y+x/2 \leqq 2$ has the resultant $0x \leqq -3$, which is clearly a contradiction. In this case we can deduce that there is no feasible solution. Unfortunately the converse is not true; it is possible to produce an example of an infeasible set of inequalities for which no loop yields a contradiction. One further idea is needed to produce a characterization.

The *closure* G' of G is the multigraph we obtain by adding to G all of the bounds that are resultants of simple cycles.

THEOREM 2.1 (Shostak [19]). *The original system G is infeasible if and only if some simple loop in the closure G' has a resultant that is a contradiction.*

3. Operations on polygons. For parallel computation we will want to be able to manipulate many bounds simultaneously. To discuss this approach it is convenient to represent a set of inequalities involving x and y by a *polygon* P_{xy} consisting of all points in the plane that satisfy all of the inequalities. (We call these objects polygons even though they may sometimes be unbounded.) It is useful to have a term to indicate

[1] The correctness of our feasibility test follows directly from the theorems of [17], but the presentation we have chosen enables us to depend only on results appearing in journals.

that membership in a polygon implies some inequality; thus we will say that a polygon *P incorporates* an inequality if all points in the polygon satisfy the inequality. The algorithm of Nelson [17] makes use of two operations on polygons, namely, intersection and composition. In this section we define these polygons and operations, and indicate how we do them efficiently in parallel. (We have not carefully optimized these algorithms since this would not significantly improve the statement of the overall complexity of our feasibility testing algorithm.) Let R denote the set of reals, and let R^{xy}, for any two variables x and y, denote the two-dimensional space $R \times R$ given by the cross product of the x and y axes.

Suppose S consists of m inequalities in n variables. For $1 \leqq k \leqq n$, let Q_{xy}^k denote the polygon corresponding to the set of all inequalities that involve no variables other than x and y, and which are resultants of paths (not necessarily simple) of length at most k. Note that Q_{xy}^1 is the polygon determined by the original inequalities involving only x and/or y. In particular, the polygon Q_{xy}^1 incorporates all of the inequalities corresponding to edges among v_0, x, and y; thus we do not need the vertex v_0 in our graphs. Clearly the polygon Q_{xx}^k incorporates the resultants of all cycles of length at most k that involve x.

The algorithm uses two basic operations on convex polygons, namely, intersection and functional composition. The *intersection* of two polygons is the ordinary set intersection of their point sets, i.e.,

$$P_{xy} \cap P'_{xy} = \{(x, y) \mid (x, y) \in P_{xy} \text{ and } (x, y) \in P'_{xy}\}.$$

The *composition* P_{zx} of two polygons P_{zy} and P_{yx} is

$$P_{zx} = P_{zy} \circ P_{yx} = \{(z, x) \mid \exists y \in R \text{ such that } (z, y) \in P_{zy} \text{ and } (y, x) \in P_{yx}\}.$$

In other words, as noted in [17], $P_{zy} \circ P_{yx}$ is the projection onto R^{zx} of the intersection of the cylinders with bases P_{zy} and P_{yx}.

We now describe the implementation of these two operations. We represent convex polygons by a domain D and two bounds L and H; D is a (possibly infinite) interval, and $L(x)$ (respectively, $H(x)$) is a convex (respectively, concave) piecewise linear function. The set of points in the represented polygon P_{yx} is

$$P_{yx} = \{(y, x) \mid x \in D \text{ and } L(x) \leqq y \leqq H(x)\};$$

for the representation to be considered valid we require that

$$x \in D \Rightarrow L(x) \leqq y \leqq H(x).$$

For simplicity we first discuss the basic operations as applied to piecewise linear convex functions rather than polygons. Thus, we first consider functions of the form $y = f(x) = \max_{1 \leqq i \leqq N} \{\alpha_i x + \beta_i\}$, which we represent by a list of pairs (α_i, β_i) ordered so that $\alpha_i < \alpha_j$ for $i < j$; we will require that there be no extraneous pairs, i.e., that no pair of values (α, β) appears more than once in the list, and that f coincides with each linear function $y = \alpha_i x + \beta_i$ over some interval of positive length.

Consider first the intersection problem. Given two functions $y = f_1(x)$ and $y = f_2(x)$ in the form described above, with N_1 and N_2 linear pieces, respectively, we have to compute the representation of $y = g(x) = \max\{f_1(x), f_2(x)\}$. This problem can be solved in $O(\log N)$ time with $O(N)$ processors, where $N = N_1 + N_2$. Here we briefly sketch the method. First note that we can convert between the representation discussed above and a list of the breakpoints (i.e., coordinates of points of discontinuity in the slope) of each function in constant time. Next, we merge the sets of breakpoints for f_1 and f_2 according to their x-coordinate, but keep track of whether each came from f_1 or f_2; call these, respectively, type 1 and type 2 breakpoints. The merging can be done

efficiently by the algorithm of [4]. Next, using standard pointer doubling techniques, each type 1 (respectively, type 2) point can determine the previous and following type 2 (respectively, type 1) point. Once this information is available, each point can determine in $O(1)$ time whether it lies below or on $g(x)$. Finally, knowing the type of its neighbors, and whether they lie below or lie on $g(x)$, each point can determine in $O(1)$ time whether f_1 and f_2 intersect between it and its neighbor. Thus we can generate a list of all breakpoints of $g(x)$. This can then be converted back to the representation, as a list of linear functions, described above.

The second operation we need for our algorithm is functional composition. We first demonstrate this operation in a special case. Suppose $y=f(x)$ and $z=g(y)$ are strictly *monotone* piecewise linear functions, each represented as described above. We would like to compute the representation of the composition $z=h(x)=g(f(x))$. Suppose f and g consist of k and l linear pieces, respectively, and let $N=k+l$. The problem can be solved by $O(N)$ processors in $O(\log N)$ time as follows. Let $y_1, \cdots, y_{l-1}$ denote the breakpoints of g. These can be found in constant time from the representation of g. Let $t_i=f^{-1}(y_i)$, $i=1, \cdots, l-1$. The t_i's can be computed in parallel in $O(\log k)$ time by a binary search. Let $x_1, \cdots, x_{k-1}$ denote the breakpoints of f. Now, the x_i's and t_i's can be merged and then the linear pieces of h can be constructed as compositions of linear functions.

Obviously, if both f and g are increasing, or if both are decreasing, then h is increasing; otherwise, h is decreasing. As for convexity or concavity properties, it is easy to verify the following:

(i) If g is monotone increasing then h is convex if both f and g are convex, and h is concave if both f and g are concave.

(ii) If g is monotone decreasing then h is convex if f is concave and g is convex, and h is concave if f is convex and g is concave.

We now sketch the construction of P_{zx} with linearly many processors in the total number of edges in P_{yx} and P_{zy}. Let y_h denote the smallest value of y (with $y \in D_{zy}$) at which $H_{zy}(y)$ attains a maximum. Note that $H_{zy}(y)$ is increasing for $y \leqq y_h$ (for $y \in D_{zy}$) and nonincreasing for $y \geqq y_h$ (again, for $y \in D_{zy}$). The function $H_{zx}(x)$ maps x to the largest value of z such that there is y in $[L_{yx}(x), H_{yx}(x)]$ for which $y \in D_{zy}$ and $L_{zy}(y) \leqq z \leqq H_{zy}(y)$. Thus, if $x \in D_{yx}$ is such that $H_{yx}(x) \leqq y_h$ (and $H_{yx}(x) \in D_{zy}$) then a least upper bound on z is obtained by setting y to $H_{yx}(x)$, that is, $H_{zx}(x) = H_{zy}(H_{yx}(x))$. On the other hand, if x is such that $L_{yx}(x) \geqq y_h$, then a least upper bound on z is obtained by setting y to $L_{yx}(x)$. Finally, if x is such that $L_{yx}(x) \leqq y_h \leqq H_{yx}(x)$, then the least upper bound on z is found by setting y to y_h. Summarizing, we have

$$H_{zx}(x) = \begin{cases} H_{zy}(L_{yx}(x)) & \text{if } y_h \leqq L_{yx}(x), \\ H_{zy}(y_h) & \text{if } L_{yx}(x) \leqq y_h \leqq H_{yx}(x), \\ H_{zy}(H_{yx}(x)) & \text{if } H_{yx}(x) \leqq y_h. \end{cases}$$

Similarly, let y_l denote the smallest value of y at which $L_{zy}(y)$ attains a minimum. Then $L_{zy}(y)$ is decreasing for $y \leqq y_l$ (with $y \in D_{zy}$) and nondecreasing for $y \geqq y_l$ (with $y \in D_{zy}$). This implies that if $x \in D_{yx}$ is such that $H_{yx}(x) \leqq y_l$, then a largest lower bound on z is obtained by picking y to be $H_{yx}(x)$, and if x is such that $L_{yx}(x) \geqq y_l$, then a largest lower bound on z is obtained by picking y to be $L_{yx}(x)$. Finally, if $L_{yx}(x) \leqq y_l \leqq H_{yx}(x)$, then we pick $y=y_l$. Thus

$$L_{zx}(x) = \begin{cases} L_{zy}(L_{yx}(x)) & \text{if } y_l \leqq L_{yx}(x), \\ L_{zy}(y_l) & \text{if } L_{yx}(x) \leqq y_l \leqq H_{yx}(x), \\ L_{zy}(H_{yx}(x)) & \text{if } H_{yx}(x) \leqq y_l. \end{cases}$$

Obviously, there exist x_{hl} and x_{hh} such that $H_{yx}(x) \geqq y_h$ if and only if $x \in [x_{hl}, x_{hh}]$. Analogously, there exist x_{ll} and x_{lh} such that $L_{yx}(x) \leqq y_l$ if and only if $x \in [x_{ll}, x_{lh}]$. Note that the values of y_l, y_h, x_{hl}, x_{hh}, x_{ll}, and x_{lh} can be found in $O(\log N)$ time. It follows that the representations of the functions $H_{zx}(x)$ and $L_{zx}(x)$ can be computed, each over at most three disjoint intervals of x, as compositions of monotone functions. Let us consider the various types of breakpoints of the functions $H_{zx}(x)$ and $L_{zx}(x)$. Obviously, any such breakpoint is of one of the following types: (i) a breakpoint of one of the functions $H_{yx}(x)$ and $L_{yx}(x)$, (ii) an inverse image under one of these functions of a breakpoint of one of the functions $H_{zy}(y)$ and $L_{zy}(y)$, (iii) one of the points x_{hl}, x_{hh}, x_{ll}, and x_{lh}. Each of the breakpoints of the functions H_{zy} (respectively, L_{zy}) contributes at most two breakpoints to H_{zx} (respectively, L_{zx}). Also, each of the breakpoints of the functions H_{yx} and L_{yx} contributes at most one breakpoint to each of the functions H_{zx} and L_{zx}. Thus, the total number of breakpoints of H_{zx} and L_{zx} is at most $2N+4$, where N is the total number of breakpoints of H_{yx}, L_{yx}, H_{zy}, and L_{zy}.

The new domain is given by

$$D_{zx} = \{x \mid x \in D_{yx} \text{ and } [L_{yx}(x), H_{yx}(x)] \cap D_{zy} \neq \varnothing\}.$$

We omit the details showing how this can be computed within the stated resource bounds.

4. Deciding feasibility. Using the basic operations of intersection and decomposition, we can now sketch the algorithm for deciding feasibility of a given system of linear inequalities with at most two variables per inequality. The algorithm consists of two iterations of the procedure UpdatePaths:

```
procedure UpdatePaths(Q);
comment Q is an array of polygons indexed by x and y;
begin
  for i ← 1 to ⌈lg n⌉ pardo
  for all variables z and x pardo
      Q_zx ← Q_zx ∩ (∩_y Q_zy ∘ Q_yx);
  POINTA: comment at this point the resultants of all paths are incorporated into
          the polygons;
  for all variables x pardo
    Q_xx ← Q_xx ∩ L_xx;
end;
```

We define I_{xx} to be the *identity polygon*, i.e., $I = \{(x, x) \mid x \in R\}$. This algorithm for deciding feasibility begins by using an algorithm analogous to the standard parallel transitive closure or shortest path algorithms (see [13] for more information about such algorithms). It is interesting to note that the two operations $\cap$ and $\circ$ do *not* form a closed semiring in the sense defined in [1]. In particular, the distributivity condition $P_{zy} \circ (P_{yx} \cap P'_{yx}) = (P_{zy} \circ P_{yx}) \cap (P_{zy} \circ P'_{yx})$ *fails* to hold in general. (As an example, let P_{zy} be the entire zy plane, let P_{yx} be determined by the one inequality $y \leqq -1$, and let P'_{yx} be determined by the one inequality $y \geqq 1$. Then the left side is the empty set and the right side is the entire zx plane.) It is not hard to see, though, that we do have

$$P_{zy} \circ (P_{yx} \cap P'_{yx}) \subseteq (P_{zy} \circ P_{yx}) \cap (P_{zy} \circ P'_{yx}).$$

From this we can easily show that, if for each x and y we initialize Q_{xy} to be the set of all inequalities involving x and y, then at POINTA we will have $Q_{xy} \subseteq Q^n_{xy}$, where

Q^n_{xy} is defined as in § 3. (Note that while we do not claim equality, the Q_{xy} do contain the projection of the feasible space onto R^{xy} since each composition and intersection corresponds to valid deductions that can be made about the feasible space.) In particular, at the end of UpdatePaths each Q_{xx} will be a set of pairs (x, x) where each x obeys the constraints added to G in § 2 to form the closure G'. Thus by Theorem 2.1, a second application of UpdatePaths will cause at least one of the Q_{xy} to become empty if the original set of inequalities had no feasible solution. Furthermore, if the feasible space is nonempty, it follows from Lemma 9 of [2] that after this second application of the procedure the projection of the feasible set onto any axis R^x is the same as the projection of Q_{xx} onto R^x.

A simplification of this description is possible: by initially restricting each Q_{xx} to be contained within I_{xx}, we eliminate the need for the last loop in UpdatePaths. In fact, then the entire feasibility checking procedure becomes the CheckFeasibility procedure:

procedure CheckFeasibility (S, V);
comment S is the set of inequalities, and V is the set of variables;
begin
  for all variables x and y **pardo**
    $Q_{xy} \leftarrow$ the polygon determined by all inequalities involving no variables outside $\{x, y\}$;
  for all variables x **pardo**
    $Q_{xx} \leftarrow Q_{xx} \cap I_{xx}$;
  for $i \leftarrow 1$ **to** $2\lceil \lg n \rceil$ **do**
    for all variables x and z **pardo**
      $Q_{zx} \leftarrow Q_{zx} \cap (\cap_y Q_{zy} \circ Q_{yx})$;
end;

This is nearly the same as the algorithm of [17], and another proof of correctness can be found there.

Let the total number of edges in all polygons constructed during the algorithm be E. As in the sequential algorithm of [17], the polygons Q_{xy} are computed in $O(\log n)$ stages, and we have $E = mn^{O(\log n)}$. Intersection of n polygons can be computed by n parallel teams of processors in $O(\log n)$ phases, where in each phase each team is computing the intersection of two polygons. It is convenient to think here of a model of computation where the machine does not have to allocate all the processors in advance; rather it invokes processors as they are needed, just like a Turing machine using unlimited tape space. This allows us to talk about the "worst-case processor complexity." Assuming we have $O(E)$ processors, all pairwise intersections and compositions take $O(\log E)$ time. It follows that the entire procedure takes $O(\log E \log^2 n)$ time. Thus, the worst-case running time is $O((\log m + \log^2 n) \log^2 n)$.

It is interesting to consider the space complexity implied by our result. We have just established that we can determine feasibility in $T = O((\log m + \log^2 n) \log^2 n)$ parallel time using $P = mn^{O(\log n)}$ processors with a concurrent-read-exclusive-write PRAM. Using standard simulation relations between parallel models of computation and between parallel time and sequential space [3], [9], [10], [13], [20], this implies that feasibility can be determined by a poly-log space-bounded deterministic Turing machine. This is strong evidence that the problem is not P-complete.

5. Computing a feasible solution. We now consider the problem of computing a feasible solution, given the projections of the (nonempty) feasible domain P onto the

individual axes. Thus, let $[x_{\text{low}}, x_{\text{high}}]$ $(-\infty \leqq x_{\text{low}} \leqq x_{\text{high}} \leqq \infty)$ denote the set of values of variable x that can be completed into a solution of the entire system S. If all the projections are *finite* intervals, then a feasible solution is readily available.

PROPOSITION 5.1. *If for every x both x_{low} and x_{high} are finite, then a feasible solution is obtained by setting each variable x to the arithmetic mean $\frac{1}{2}(x_{\text{low}} + x_{\text{high}})$.*

Proof. Suppose, to the contrary, that the vector whose components are the arithmetic means $\frac{1}{2}(x_{\text{low}} + x_{\text{high}})$ is not feasible. Then there is an inequality $\alpha x + \beta y \leqq \gamma$ that is violated. In other words,

$$\tfrac{1}{2}\alpha(x_{\text{low}} + x_{\text{high}}) + \tfrac{1}{2}\beta(y_{\text{low}} + y_{\text{high}}) > \gamma.$$

Consider the rectangle $[x_{\text{low}}, x_{\text{high}}] \times [y_{\text{low}}, y_{\text{high}}]$. By definition, each edge of this rectangle contains at least one point of the projection P_{xy}. However, we claim that this contradicts the inequality

$$\tfrac{1}{2}\alpha(x_{\text{low}} + x_{\text{high}}) + \tfrac{1}{2}\beta(y_{\text{low}} + y_{\text{high}}) > \gamma,$$

since the center of the rectangle is in the convex hull of any set that intersects all four edges of the rectangle. The proof of this claim is easy. Let L, R, T, and B denote points (not necessarily distinct) that lie on the left, right, top, and bottom edges of the rectangle, respectively. Consider the straight line determined by the points L and R. If the center lies on this line then we are done. Otherwise, if the center lies above the line then it is in the triangle determined by T together with L with R, and if it lies below this line then it is in the triangle determined by B together with L and R. □

Interestingly, Proposition 5.1 does not hold if there are more than two variables per inequality. To see this, consider the system $x \geqq 0$, $y \geqq 0$, $z \geqq 0$, and $x + y + z \leqq 1$. The projection of the feasible space onto the x-, y-, or z-axis is just $[0, 1]$, but the point $(\frac{1}{2}, \frac{1}{2}, \frac{1}{2})$ is not feasible.

The unbounded case is handled as follows. We introduce to the system an additional variable ξ and the $2n$ inequalities $x_j \leqq \xi$, $x_j \geqq -\xi$ $(j = 1, \cdots, n)$. We find the projection of the augmented problem onto the ξ-axis. In other words, we compute an interval $I = [\xi_{\text{low}}, \xi_{\text{high}}]$ $(0 \leqq \xi_{\text{low}} \leqq \xi_{\text{high}} = \infty)$, such that for every $\xi \in I$ there exist values for $x_1, \cdots, x_n$ which solve the augmented problem. By setting ξ to any finite number in I we obtain a feasible system of linear inequalities (with at most two variables per inequality) whose set of solutions is bounded. Any solution of the latter yields a solution to the original problem simply by dropping ξ. Thus we have the following proposition.

PROPOSITION 5.2. *If a system of linear inequalities has a nonempty set of solutions, then a solution can be found in poly-log time with $mn^{O(\log n)}$ processors in the worst case.*

6. Optimization problems. We have already shown that, with a general objective function, the optimization problem with at most two variables per inequality is P-complete. In this section we discuss the case where the objective function also has at most two variables with nonzero coefficients.

Intuitively, the optimization problem can be solved by searching for the optimum value, using the feasibility checking algorithm as an "oracle." In the context of sequential computation this yields a polynomial-time (but not strongly polynomial-time) algorithm. In the context of parallel computation this approach does not provide a poly-log algorithm since the number of queries during the search is *linear* in the length of the binary representation of the input.

Here we can use a technique presented in [16] to obtain a poly-log algorithm for finding optimum solutions over any ordered field. Here is a sketch of the method; see

[16] for more details. Suppose the problem is to minimize the function

$$f(x_1, \cdots, x_n) = c_1 x_1 + c_2 x_2$$

subject to a system S of linear inequalities in $x_1, \cdots, x_n$ with at most two variables per inequality. Consider the system S' of inequalities, which is obtained by adding to S an inequality $c_1 x_1 + c_2 x_2 \leqq \lambda$, where λ is a parameter. We need to find the smallest value of λ for which S' is feasible. Denote this optimum value by λ^*. We can run our parallel algorithm for deciding feasibility on S', handling λ as an indeterminate. Thus the "program variables" will be functions of λ rather than field elements. Throughout the execution of the algorithm we maintain an interval of values of λ, guaranteed to contain λ^*, over which the current program variables are all linear functions of λ. Comparisons between two functions of λ must be resolved according to the function values at λ^*, which is itself not known. However, during each step of the algorithm, each processor that is unable to perform a comparison for which it is responsible simply reports the value of λ which is critical for that comparison, that is, a value λ' such that the comparison between the two functions can be resolved by comparing λ' and λ^*. The comparison between λ' and λ^* can be carried out by setting λ to λ' and checking feasibility of the system. Let p denote a bound on the number of processors required to check feasibility. For the parametric algorithm we can either use p^2 processors, in which case all the critical values of λ can be tested in parallel, or only p processors and run a binary search over the set of critical values. In either case we obtain a poly-log algorithm with $mn^{O(\log n)}$ processors for computing λ^* over any ordered field.

Once λ^* is known, we can *solve* the system S' with $\lambda = \lambda^*$.

7. Bounded tree-width. Robertson and Seymour [18] introduced the notion of the *tree-width* of a graph. This notion lends itself via the constraints graph to systems of linear inequalities with at most two variables per inequality.

DEFINITION 7.1. A connected graph G is said to have *tree-width* less than or equal to k if there is a family $\mathbf{V} = \{V_1, \cdots, V_t\}$ of sets V_i of vertices of G with the following properties:

(i) Each V_i contains at most $k+1$ vertices of G.

(ii) For every edge e of G, there exists an i such that e has both its endpoints lying in V_i.

(iii) The intersection graph $\mathbf{T} = (\mathbf{V}, \mathbf{E})$, where $(V_i, V_j) \in \mathbf{E}$ if and only if $V_i \cap V_j \neq \varnothing$, is a tree.

We assume the graph is given together with such a tree and develop an algorithm that relies on the tree. Note that a tree with at most n nodes suffices. It will follow that if the tree-width is bounded, then the number of edges remains polynomial in m and n during the execution of the special algorithm.

For our purpose here we may assume, without loss of generality, that our graphs are connected. Also, for simplicity of presentation, assume all the sets V_i are $(k+1)$-cliques in G; this assumption is also made without loss of generality since redundant inequalities can always be added to the system.

PROPOSITION 7.2. *Suppose U, V, and W are nodes of $\mathbf{T}$ such that V lies on the path connecting U and W. Let $u \in U$ and $w \in W$ be vertices of G. Then on any path in G connecting u and w there is at least one vertex $v \in V$.*

Proof. Consider any such path $u = v_1, \cdots, v_r = w$. For every i, $i = 1, \cdots, r-1$, there is a set $V_i \in \mathbf{T}$ such that both v_i and v_{i+1} are in V_i. By definition each (V_i, V_{i+1}) is an arc in $\mathbf{T}$ (if $V_i \neq V_{i+1}$). Thus, $V_1, \cdots, V_{i-1}$ yields a path in $\mathbf{T}$. It follows that one of the V_i's equals V. This implies that one of the v_i's is in V. □

Given the underlying tree **T**, we can decompose the graph G in an efficient way. The decomposition is based on the *centroid* which is often useful in the design of parallel algorithms (see [5]). The centroid of a tree T with N nodes is a node c so that there exist two subtrees T_1, T_2 rooted at c (and also c is the only common node), each with no more than $(2N/3)+1$ nodes, whose union is T. The *centroid decomposition* of a tree is the iterated partitioning of a tree in this way into two subtrees rooted at the centroid. This decomposition is obtained in $O(\log N)$ iterations, and moreover, it can be computed in poly-log time with a polynomial number of processors [5].

In view of Proposition 7.2 the centroid decomposition of **T** induces a decomposition of G as follows. At the first level of the decomposition we have a set C of $k+1$ vertices of G and two induced subgraphs G_1, G_2, whose vertex sets intersect at C and cover all the vertices of G. Moreover, every edge of G is contained in one of these two graphs. The decomposition is iterated until all the subgraphs consist of not more than $k+1$ vertices. It follows that this decomposition has only $O(\log n)$ levels.

Given the decomposition of G, we produce polygons $Q_{xy}(G)$ as follows. The polygon Q_{xy} computed will incorporate all of the resultants of *simple* paths from x to y. (Recall that a simple path can begin and end at the same point, so x and y may be equal.) Let G_1, G_2, and C be as explained above. We state the algorithm recursively. Thus, assume we have computed polygons $Q_{xy}(G_i)$ for all pairs of vertices $x, y \in G_i$ $(i=1,2)$. In particular, if $x, y \in C$ then we have for them *two* polygons $Q_{xy}(G_1)$ and $Q_{xy}(G_2)$.

The recursive step is performed as follows. Let x and y be any two vertices of G for which we compute $Q_{xy}(G)$. For simplicity of notation assume without loss of generality that $x \in G_1$. Any simple path from x to y can be represented as a union of paths $\pi(z_0, z_1), \pi(z_1, z_2), \cdots, \pi(z_{l-1}, z_l)$ where $z_0 = x$, $z_l = y$, $z_i \in C$ for $i = 1, \cdots, l-1$, and $\pi(z, z')$ denotes some simple path from z to z'. Moreover, paths of the form $\pi(z_{2i}, z_{2i+1})$ stay entirely within G_1 while paths of the form $\pi(z_{2i-1}, z_{2i})$ stay entirely within G_2. Thus, to incorporate the resultants of all simple paths connecting x and y in G, it suffices to intersect all the polygons obtained by compositions of the form

$$Q_{xz_1}(G_1) \circ Q_{z_1z_2}(G_2) \circ Q_{z_2z_3}(G_1) \circ \cdots \circ Q_{z_{l-1}y}(G_i)$$

(where $y \in G_i$), so that the z_j's $(1 \leqq j \leqq l-1)$ are pairwise distinct points in C. The number of different choices of the z_j's implies that for each pair x, y, the number of polygons intersected this way is bounded by a constant K depending only on k. For each pair x and y, each composition is of at most $k+2$ polygons. This may multiply the number of breakpoints by at most $O(k)$, since composition of k polygons can be computed in $O(\log k)$ compositions of two polygons (where the number of breakpoints is at most approximately doubled). Since the entire process runs in $O(\log n)$ stages, and there are m inequalities at the beginning, it follows that the number of edges in each of the generated polygons is $m(kK)^{O(\log n)}$. This is the same as $mn^{g(k)}$ for some $g(k)$ depending only on k; hence it is polynomial in m and n for any fixed k. The running time on a suitable number of processors is $O(\log^2 n \log m)$ with a coefficient that depends on k. By Theorem 2.1 this algorithm can determine feasibility.

8. Directions for further work. It is interesting to ask whether the algorithms we have described in §§ 4–6 can ever in fact require more than polynomially many processors. This is essentially the same as the question asked in [17] of whether the algorithm of [17] can require more than polynomial time.

More generally, resolving whether the linear programming problem with two variables per inequality lies in NC seems like a very interesting question. To provide context, note that Cook fairly recently observed [6, p. 18] "I find it interesting that

very few natural problems in [poly-log space] have come to my attention which are not in NC. One notable exception is the problem of determining whether two groups, presented by their multiplication tables, are isomorphic.... I know of no NC solution to this problem, or even any polynomial time solution." Thus the present status of linear programming with two variables per inequality seems to be rather unusual, particularly since it is known to be solvable in polynomial time (even if we allow that inputs are arbitrary reals and the time bound must be independent of these values [15]).

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] B. ASPVALL AND Y. SHILOACH, *A polynomial time algorithm for solving systems of linear inequalities with two variables per inequality*, SIAM J. Comput., 9 (1980), pp. 827–845.

[3] A. BORODIN, *On relating time and space to size and depth*, SIAM J. Comput., 6 (1977), pp. 733–744.

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

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

[6] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.

[7] D. DOBKIN, R. J. LIPTON, AND S. REISS, *Linear programming is log space hard for P*, Inform. Process. Lett., 8 (1979), pp. 96–97.

[8] L. R. FORD, JR. AND R. D. FULKERSON, *Flows in Networks*, Princeton University Press, Princeton, NJ, 1962.

[9] S. FORTUNE AND J. WYLLIE, *Parallelism in random access machines*, in Proc. 10th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1978, pp. 114–118.

[10] L. M. GOLDSCHLAGER, *Synchronous parallel computation*, Tech. Report 114, Department of Computer Science, University of Toronto, Toronto, Ontario, Canada, December 1977.

[11] L. M. GOLDSCHLAGER, R. A. SHAW, AND J. STAPLES, *The maximum flow problem is log space complete for P*, Theoret. Comput. Sci., 21 (1982), pp. 105–111.

[12] D. HELMBOLD AND E. MAYR, *Fast scheduling problems on parallel computers*, Report STAN-CS-84-1025, Computer Science Department, Stanford University, Stanford, CA, 1984.

[13] R. M. KARP AND V. RAMACHANDRAN, *Parallel algorithms for shared memory machines*, in Handbook of Theoretical Computer Science, J. van Leeuwen, ed., North-Holland, Amsterdam, 1988.

[14] L. G. KHACHIYAN, *A polynomial algorithm in linear programming*, Soviet Math. Dokl., 20 (1979), pp. 191–194.

[15] N. MEGIDDO, *Towards a genuinely polynomial algorithm for linear programming*, SIAM J. Comput., 12 (1983), pp. 347–353.

[16] ———, *Applying parallel computation algorithms in the design of serial algorithms*, J. Assoc. Comput. Mach., 30 (1983), pp. 852–865.

[17] C. G. NELSON, *An $n^{O(\log n)}$ algorithm for the two-variable-per-constraint linear programming satisfiability problem*, Report STAN-CS-76-689, Department of Computer Science, Stanford University, Stanford, CA, November 1978.

[18] N. ROBERTSON AND P. D. SEYMOUR, *Graph width and well-quasi-ordering: a survey*, in Progress in Graph Theory, Academic Press, Canada, 1984, pp. 399–406.

[19] R. SHOSTAK, *Deciding linear inequalities by computing loop residues*, J. Assoc. Comput. Mach., 28 (1981), pp. 769–779.

[20] L. STOCKMEYER AND U. VISHKIN, *Simulation of parallel random access machines by circuits*, SIAM J. Comput., 13 (1984), pp. 409–422.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1011–1023, December 1990

A TIME COMPLEXITY GAP FOR TWO-WAY PROBABILISTIC FINITE-STATE AUTOMATA*

CYNTHIA DWORK† AND LARRY STOCKMEYER†

Abstract. It is shown that if a two-way probabilistic finite-state automaton (2pfa) M recognizes a nonregular language L with error probability bounded below $\frac{1}{2}$, then there is a positive constant b (depending on M) such that, for infinitely many inputs x, the expected running time of M on input x must exceed 2^{n^b} where n is the length of x. This complements a result of Freivalds showing that 2pfa's can recognize certain nonregular languages in exponential expected time. It also establishes a time complexity gap for 2pfa's, since any regular language can be recognized by some 2pfa in linear time. Other results give roughly exponential upper and lower bounds on the worst-case increase in the number of states when converting a polynomial-time 2pfa to an equivalent two-way nondeterministic finite-state automaton or to an equivalent one-way deterministic finite-state automaton.

Key words. finite state automata, probabilistic automata, complexity theory

AMS(MOS) subject classifications. 68Q15, 68Q75

1. Introduction.

The power of randomization is a recurring theme in the theory of computation. One of the fundamental open questions of complexity theory asks whether the class BPP of languages recognizable in polynomial time by probabilistic Turing machines is larger than the class P of languages recognizable in polynomial time by deterministic Turing machines. To gain insight into the power of randomization, some research has focused on more restricted models of computation. For example, Freivalds [2] and Gill [4] have shown that probabilistic one-tape Turing machines can recognize certain languages more efficiently than deterministic one-tape Turing machines.

Another example of the power of randomization, which provided the motivation for this paper, concerns two-way probabilistic finite-state automata (2pfa's). A 2pfa consists of a probabilistic finite-state control and an input tape which is scanned by a single two-way head, that is, the head can move both left and right (a complete definition appears in § 2). Freivalds [3] has shown that, for any $\varepsilon > 0$, there is a 2pfa which recognizes the nonregular language $L_0 = \{0^m 1^m \mid m \geqq 1\}$ with error probability at most ε. In contrast, it is known that deterministic (and even nondeterministic and alternating) two-way finite-state automata can recognize only regular languages [8], [13], [15]. A property of Freivald's result is that the 2pfa constructed to recognize L_0 uses exponential expected time. In the original construction in [3], the expected time is proportional to $n2^n$, where n is the length of the input. The construction can be easily modified to give, for any constant $c > 0$, a 2pfa recognizing L_0 in expected time $O(2^{cn})$. Greenberg and Weiss [5] show that this expected time bound cannot be improved further. They show that no 2pfa running in expected time $2^{o(n)}$ can recognize L_0 with error probability bounded below $\frac{1}{2}$. This raises the question of whether there is some other nonregular language which a 2pfa can recognize efficiently, for example, in polynomial expected time. The main result of this paper answers this question negatively: if a 2pfa M recognizes a nonregular language with error probability bounded below $\frac{1}{2}$, then there is a constant $b > 0$ such that, for infinitely many n, the expected

* Received by the editors May 8, 1989; accepted for publication (in revised form) December 14, 1989. An extended abstract containing some of the results in this paper appeared in the Proceedings of the 30th IEEE Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, 1989.

† IBM Research Division, K53/802, 650 Harry Road, San Jose, California 95120.

running time of M must exceed 2^{n^b}. Thus, although randomization adds power to two-way finite-state automata, this added power comes with an added cost in expected running time. The result can also be viewed as a gap theorem for 2pfa's: for any language L, either L can be recognized in time $O(n)$ (if L is regular) or L requires time 2^{n^b} infinitely often (if L is not regular).

A tool in the proof of the main result is a measure $N_L(n)$ of the nonregularity of a language L. Intuitively, when L is restricted to words of length at most n, $N_L(n)$ measures the number of words which must be "distinguished" by any recognizer of L, where words w and w' must be distinguished on the restricted L if there exists a word v such that the lengths of wv and $w'v$ are both at most n, and $wv \in L$ if and only if $w'v \notin L$. In § 3 we show that if L is not regular then $N_L(n) \geqq \sqrt{n} - 1$ for infinitely many n. In § 4 we show that, if a 2pfa M recognizes L in expected time $T(n)$, then a lower bound on $T(n)$ can be given in terms of $N_L(n)$. Combining this with the lower bound on $N_L(n)$ for any nonregular L, the main result follows easily.

In the final three sections we mention some related results. In § 5 the main result is extended to show that a probabilistic Turing machine with space bound $o(\log \log n)$ which runs in polynomial expected time cannot recognize any nonregular language with error probability bounded below $\frac{1}{2}$. In § 6 we consider conciseness issues, that is, the relative number of states required by probabilistic versus deterministic or nondeterministic finite-state machines to recognize the same regular language. When 2pfa's are restricted to halt in polynomial expected time, 2pfa's are at most exponentially more concise than one-way deterministic finite-state automata, and there are examples showing that 2pfa's can be exponentially more concise than two-way nondeterministic finite state automata. In § 7 we note that the main result is not true if the error probability is not bounded below $\frac{1}{2}$.

2. Definitions. A particular 2pfa M is specified by a finite set Q of states, a finite input alphabet Σ, and a transition function δ. The set Q contains designated states q_0 (the initial state), q_a (the accepting state), and q_r (the rejecting state). Let $\not\epsilon$ be a symbol not in Σ. The transition function has the form

$$\delta : (Q - \{q_a, q_r\}) \times (\Sigma \cup \{\not\epsilon\}) \times Q \times \{\text{left, right, stationary}\} \to [0, 1],$$

where, for each fixed q and σ, the sum of $\delta(q, \sigma, q', d)$ over all q' and d equals one. The meaning of δ is that, if M is in state q with the head scanning the symbol σ, then with probability $\delta(q, \sigma, q', d)$ the machine enters state q' and either moves the head one symbol in direction d if $d \in \{\text{left, right}\}$ or does not move the head if $d = \text{stationary}$. The computation of M on input $x \in \Sigma^*$ begins with the word $\not\epsilon x \not\epsilon$ written on the input tape; the head is positioned on the left endmarker $\not\epsilon$, and the state is q_0. The computation is then governed (probabilistically) by the transition function δ until M either accepts by entering state q_a or rejects by entering state q_r. We assume that δ is defined so that the head never moves outside the word $\not\epsilon x \not\epsilon$. *$M$ halts* when it enters state q_a or q_r.

Let $L \subseteq \Sigma^*$, let M be a 2pfa with input alphabet Σ, and let $0 \leqq \varepsilon < \frac{1}{2}$. Then *$M$ recognizes L within error probability ε* if

1) For all $x \in L$, $\Pr[M \text{ accepts } x] \geqq 1 - \varepsilon$, and
2) For all $x \notin L$, $\Pr[M \text{ rejects } x] \geqq 1 - \varepsilon$.

M recognizes L if M recognizes L within error probability ε for some $\varepsilon < \frac{1}{2}$. One reason to bound the error probability below $\frac{1}{2}$ is so that the error probability can be significantly decreased by running the machine several times on the same input and taking the majority answer. Specifically, if a 2pfa M recognizes L and halts with probability one on all inputs, then for any $\varepsilon > 0$ there is a 2pfa M' which recognizes L within error probability ε.

Although our lower bound result allows δ to take arbitrary real values in $[0, 1]$, other results construct particular 2pfa's which need only a fair coin to make random choices. A 2pfa is a *coin-tossing* 2pfa if the range of its transition function δ is $\{0, \frac{1}{2}, 1\}$. For example, the result of Freivalds [3] mentioned above is true for a coin-tossing 2pfa. (It is not hard to see that 2pfa's with rational transition probabilities are no more powerful than coin-tossing 2pfa's, although we do not know whether 2pfa's with real transition probabilities are more powerful than coin-tossing 2pfa's.)

Let $|x|$ denote the length of the word x. The *expected running time* of the 2pfa M on the input x is the expected number of steps in the computation of M on input x, where the expectation is taken over the random choices made by M. The 2pfa M *runs within expected time* $T(n)$ if, for all n and all inputs x with $|x| \leqq n$, the expected running time of M on x is at most $T(n)$.

Logarithms in this paper are to the base two.

3. A measure of nonregularity. Let $L \subseteq \Sigma^*$ and let n be a positive integer. We first define a relation $\sim_{L,n}$ on words. For $w, w' \in \Sigma^*$ with $|w| \leqq n$ and $|w'| \leqq n$, the words w and w' are *n-similar*, written $w \sim_{L,n} w'$, if, for all $v \in \Sigma^*$ such that $|wv| \leqq n$ and $|w'v| \leqq n$, we have $wv \in L$ if and only if $w'v \in L$. The words w and w' are *n-dissimilar*, written $w \not\sim_{L,n} w'$, if $|w| \leqq n$, $|w'| \leqq n$, and it is not the case that $w \sim_{L,n} w'$, that is, there exists a v with $|wv| \leqq n$, $|w'v| \leqq n$, and $wv \in L$ if and only if $w'v \notin L$. Note that if $w \not\sim_{L,n} w'$, then w and w' must be "distinguished" as described in the Introduction. The relations $\sim_{L,n}$ and $\not\sim_{L,n}$ are not defined for words of length greater than n. We remark that $\sim_{L,n}$ is reflexive and symmetric, although it is not transitive in general.

Let $N_L(n)$ be the maximum k such that there exist k distinct words $w_1, \cdots, w_k$ which are pairwise $\not\sim_{L,n}$, that is, $w_i \not\sim_{L,n} w_j$ for all $1 \leqq i < j \leqq k$. It is clear that $N_L(n)$ is nondecreasing in n.

LEMMA 3.1. *$N_L(n)$ is bounded above by a constant if and only if L is regular.*

Proof. For words w and w', define wR_Lw' if, for all words v, we have $wv \in L$ if and only if $w'v \in L$. R_L is an equivalence relation. By the Myhill–Nerode theorem (see [7]), R_L has finite index if and only if L is regular.

If L is regular, it is obvious from definitions that $N_L(n)$ is bounded above by the (finite) index of R_L. Conversely, if L is not regular, then, for any $k \geqq 2$ there exist words $w_1, \cdots, w_k$ and words v_{ij} such that $w_iv_{ij} \in L$ if and only if $w_jv_{ij} \notin L$ for all $1 \leqq i < j \leqq k$. Choosing n_0 to be the maximum length of the words w_iv_{ij}, w_jv_{ij} over all such i and j, we have $N_L(n_0) \geqq k$. Since k was arbitrary, $N_L(n)$ is unbounded. □

For the remaining four lemmas in this section, fix some language L, and abbreviate $\sim_{L,n}$ and $\not\sim_{L,n}$ by $\sim_n$ and $\not\sim_n$, respectively. The next two lemmas establish two useful properties of $\sim_n$ and $\not\sim_n$.

LEMMA 3.2. *Let w, w', v be words such that $w \sim_n w'$, $|wv| \leqq n$, and $|w'v| \leqq n$. Then $wv \sim_n w'v$.*

Proof. Suppose to the contrary that $wv \not\sim_n w'v$. Therefore, there exists a word z such that $|wvz| \leqq n$, $|w'vz| \leqq n$, and $wvz \in L$ if and only if $w'vz \notin L$. But now vz is a witness to $w \not\sim_n w'$, contradicting the assumption that $w \sim_n w'$. □

LEMMA 3.3. *Let w, w', s be words such that $|s| \leqq |w|$, $w \sim_n s$, and $w \not\sim_n w'$. Then $s \not\sim_n w'$.*

Proof. Let v be such that $|wv| \leqq n$, $|w'v| \leqq n$, and $wv \in L$ if and only if $w'v \notin L$. We have $|sv| \leqq |wv| \leqq n$. Since $w \sim_n s$, we have $sv \in L$ if and only if $wv \in L$. Therefore, $sv \in L$ if and only if $w'v \notin L$, so $s \not\sim_n w'$. □

The next lemma shows that we can always find $N_L(n)$ "short" words which are pairwise $\not\sim_n$.

LEMMA 3.4. *Let $k = N_L(n)$. There exist words $s_1, \cdots, s_k$ which are pairwise $\not\sim_n$ and $|s_i| \leqq k$ for all i.*

Proof. By definition, there are words $w_1, \cdots, w_k$ which are pairwise $\not\sim_n$. If $|w_i| \leqq k$ for all i, we are done. Suppose that $|w_i| > k$ for some i. Consider the $|w_i|$ nonempty prefixes of w_i. Since $N_L(n) = k$, these prefixes cannot be pairwise $\not\sim_n$. Therefore, we can write $w_i = xyz$ where $|y| \geqq 1$ and $x \sim_n xy$. Since $|xz| < |xyz| \leqq n$, it follows from Lemma 3.2 that $xz \sim_n xyz = w_i$. For an arbitrary j with $1 \leqq j \leqq k$ and $j \neq i$, we have $w_i \not\sim_n w_j$ by choice of the w's. Taking $s = xz$ in Lemma 3.3, we see that $xz \not\sim_n w_j$. Replacing w_i by xz in the list $w_1, \cdots, w_k$, we obtain a new set of pairwise $\not\sim_n$ words where the length of w_i has been reduced. Continuing in this way, k pairwise $\not\sim_n$ words of length at most k can be constructed. □

LEMMA 3.5. *Let n and k be such that $N_L(n-1) < N_L(n) = k$. Then $n \leqq 2k + k^2$.*

Proof. Assume to the contrary that $n > 2k + k^2$. By Lemma 3.4, there are pairwise $\not\sim_n$ words $s_1, \cdots, s_k$ with $|s_i| \leqq k$ for all i. Our goal is to show that, for all i and j with $1 \leqq i < j \leqq k$, there exists a word u such that $|s_i u| \leqq n-1$, $|s_j u| \leqq n-1$, and $s_i u \in L$ if and only if $s_j u \notin L$. This implies that $s_1, \cdots, s_k$ are pairwise $\not\sim_{n-1}$, contradicting the assumption that $N_L(n-1) < k$.

Since $s_i \not\sim_n s_j$, there is a word v with $|s_i v| \leqq n$, $|s_j v| \leqq n$, and $s_i v \in L$ if and only if $s_j v \notin L$. If $|s_i v|$ and $|s_j v|$ are both less than n, we are done. So assume (without loss of generality) that $|s_i v| = n$. Let $v_1, \cdots, v_m$ be the prefixes of v having length at least k. Since $|s_i| \leqq k$ and $n > 2k + k^2$, it follows that $m > k^2$. For each word $s_i v_l$ with $1 \leqq l \leqq m$ there exists an a such that $s_i v_l \sim_n s_a$, for otherwise there would be $k+1$ pairwise $\not\sim_n$ words, contradicting the assumption that $k = N_L(n)$. Similarly, for each l there is a b such that $s_j v_l \sim_n s_b$. Since there are more than k^2 choices for l, and exactly k^2 choices of pairs (a, b), it follows that there exist numbers a, b and words x, y, z such that $v = xyz$, $|x| \geqq k$, $|y| \geqq 1$, and

$$s_i x \sim_n s_a, \tag{1}$$

$$s_i xy \sim_n s_a, \tag{2}$$

$$s_j x \sim_n s_b, \tag{3}$$

$$s_j xy \sim_n s_b. \tag{4}$$

Since $|v| \leqq n$ and $|x| \geqq k$, we have $|z| \leqq n-k$, so $|s_a z| \leqq k + (n-k) = n$. Similarly, $|s_b z| \leqq n$. Finally,

$$\begin{aligned}
s_i xz \in L \quad & \text{iff } s_a z \in L && \text{by (1)} \\
& \text{iff } s_i xyz \in L && \text{by (2)} \\
& \text{iff } s_i v \in L && \text{since } v = xyz \\
& \text{iff } s_j v \notin L && \text{by choice of } v \\
& \text{iff } s_j xyz \notin L && \text{since } v = xyz \\
& \text{iff } s_b z \notin L && \text{by (4)} \\
& \text{iff } s_j xz \notin L && \text{by (3).}
\end{aligned}$$

Therefore, $u = xz$ is a word such that $|s_i u| \leqq n-1$, $|s_j u| \leqq n-1$, and $s_i u \in L$ if and only if $s_j u \notin L$. □

We can now obtain our main result about $N_L(n)$.

THEOREM 3.6. *If L is not regular, then $N_L(n) \geqq \sqrt{n} - 1$ for infinitely many n.*

Proof. By Lemma 3.1, there are infinitely many n such that $N_L(n-1) < N_L(n)$. For each such n, we have $N_L(n) \geqq \sqrt{n} - 1$ by Lemma 3.5. □

Two remarks about strengthening Theorem 3.6 can be made. First, the lower bound $\sqrt{n}-1$ could be improved to at most $n+1$, since if $L \subseteq \{0\}^*$ then $N_L(n) \leqq n+1$ for all n. We have not pursued such an improvement, since it would not improve the lower bound on expected time obtained in the next section. Second, regarding that the lower bound has been shown to hold only for infinitely many n, no interesting lower bound on $N_L(n)$ can be proved for almost all n, given only that L is not regular. If $g(n)$ is any unbounded function, there is a nonregular L such that $N_L(n) \leqq g(n)$ for infinitely many n. For integers $0 = n_1 < n_2 < n_3 < \cdots$, define

$$L = \{0^m \mid (\exists i)[n_{i-1} < m \leqq n_i \text{ and } m \equiv 0 \bmod i]\}.$$

Then $N_L(n_i) \leqq n_{i-1} + 1 + i$ for all $i \geqq 2$. So by choosing the n_i to be far enough apart, we can satisfy $N_L(n_i) \leqq g(n_i)$ for all $i \geqq 2$.

4. A lower bound on expected time. In this section we give a lower bound, in terms of $N_L(n)$, on the expected time required by a 2pfa to recognize L. The proof method is one which we used in [1]; it is similar to methods used previously by Rabin [12] and Greenberg and Weiss [5].

Since we model computations of 2pfa's by Markov chains, we first give some definitions and results about Markov chains. Basic facts about Markov chains can be found, for example, in [14]. We consider Markov chains with finite state space $\{1, 2, \cdots, m\}$ for some m. A particular Markov chain is completely specified by its matrix $P = \{p_{ij}\}_{i,j=1}^{m}$ of transition probabilities. If the Markov chain is in state i, then it next moves to state j with probability p_{ij}. The chains we consider have a designated starting state, state 1, and two absorbing states, states $m-1$ and m (so $p_{m-1,m-1} = p_{m,m} = 1$). States other than $m-1$ and m are either transient or inaccessible from state 1. Let $a(P)$ denote the probability that the Markov chain P is absorbed in state m when started in state 1. Let $t(P)$ denote the expected time to absorption, meaning absorption into one of the states $m-1$ or m.

We are concerned with how the probability $a(P)$ is affected by small changes to the transition probabilities. Let $\beta \geqq 1$. Say that two numbers p and p' are *β-close* if either (i) $p = p' = 0$ or (ii) $p > 0$, $p' > 0$, and $\beta^{-1} \leqq p/p' \leqq \beta$. Two Markov chains $P = \{p_{ij}\}_{i,j=1}^{m}$ and $P' = \{p'_{ij}\}_{i,j=1}^{m}$ are *β-close* if, for all i and j, p_{ij} and p'_{ij} are β-close.

As noted by Greenberg and Weiss [5] (see also [1, Lem. 3.2]), the following lemma can be proved easily from the Markov chain tree theorem [9], [10].

LEMMA 4.1. *Let P and P' be two m-state Markov chains which are β-close. Then $a(P)$ and $a(P')$ are β^{2m}-close.*

We need a variation of this lemma where certain corresponding pairs of probabilities p_{ij} and p'_{ij} are not known to be β-close, but only in the case that these probabilities are both much smaller than the reciprocal of the expected time to absorption. For $\beta \geqq 1$ and $\lambda \geqq 0$, we say that P and P' are *β-close* mod λ if, for each pair i, j, either

1) $p_{ij} \leqq \lambda$ and $p'_{ij} \leqq \lambda$, or

2) $p_{ij} > \lambda$, $p'_{ij} > \lambda$, and p_{ij} and p'_{ij} are β-close.

The basic idea is that, if λ is much smaller than both $t(P)^{-1}$ and $t(P')^{-1}$, then a transition from state i to state j is unlikely to occur (before absorption) in case 1), so such transitions can be essentially ignored. The reader satisfied with this intuition might want to skip the proof of the next lemma.

LEMMA 4.2. *Let P and P' be two m-state Markov chains which are β-close* mod λ. *Let $t = \max\{t(P), t(P')\}$. Then*

$$a(P') \geqq (1 - 2\lambda m^3)\beta^{-2m} a(P) - 4\sqrt{\lambda m t}.$$

Proof. Assume $\lambda \leqq m^{-3}$, since the lemma clearly holds otherwise. Say that a number p is *small* if $p \leqq \lambda$. We transform P and P' to Markov chains R and R', respectively, by changing all small transition probabilities to zero and altering the nonsmall probabilities only slightly. We can then bound $a(P)$ in terms of $a(R)$, similarly bound $a(P')$ in terms of $a(R')$, and apply Lemma 4.1 to R and R'.

In order to define certain events, we first describe a particular way of "running" the Markov chains P and P'. Since the description is essentially identical for P and P', we focus on P. For each state i, order the transitions from state i so that all small transition probabilities precede all nonsmall transition probabilities and so that the first nonsmall transition probability is at least $1/m$. More precisely, for each state i, choose a permutation π_i of $\{1, 2, \cdots, m\}$ and an integer l_i with $1 \leqq l_i \leqq m$ such that $p_{i,\pi_i(k)}$ is small if and only if $k < l_i$, and $p_{i,\pi_i(l_i)} \geqq 1/m$. We say that $\pi_i(l_i)$ is *special* for i. Let S be a random variable which takes a real value uniformly distributed in $[0, 1]$. We use S to run P as follows. If P is in state i, call S to obtain a random value v. Then P next enters state $\pi_i(z)$ where

$$\sum_{k=1}^{z-1} p_{i,\pi_i(k)} < v \leqq \sum_{k=1}^{z} p_{i,\pi_i(k)}.$$

So for each z, P next enters state $\pi_i(z)$ with probability $p_{i,\pi_i(z)}$.

Let $\mathscr{E}$ be the event that P is absorbed into state m, and let $\mathscr{F}$ be the event that P is absorbed (into state m or $m-1$) before any call of S produces a value v with $v \leqq \lambda m$. Note that if $\mathscr{F}$ holds, then no transition $i \to j$ with small p_{ij} is taken before absorption. Let $\mathscr{E}'$ and $\mathscr{F}'$ be the analogous events for P'. Let A be the random variable giving the number of steps of P before absorption. Let $b = (\lambda m t)^{-1/2}$. Let $\bar{\mathscr{F}}$ be the complement of $\mathscr{F}$. Since $E(A) = t(P) \leqq t$,

$$\Pr[A > bt] \leqq b^{-1}.$$

By definition of $\mathscr{F}$, and using the inequality $(1-x)^y \geqq 1 - xy$ which is valid for all real x and y with $0 \leqq x \leqq 1$ and $y \geqq 1$ [6, Thm. 42],

$$\Pr[\bar{\mathscr{F}} \mid A \leqq bt] \leqq 1 - (1-\lambda m)^{bt} \leqq \lambda m b t.$$

Therefore,

$$\Pr[\bar{\mathscr{F}}] \leqq b^{-1} + \lambda m b t = 2\sqrt{\lambda m t}. \tag{5}$$

Note that $a(P) = \Pr[\mathscr{E}]$ by definition. We will soon define the Markov chains R and R' so that $a(R) = \Pr[\mathscr{E} \mid \mathscr{F}]$ and $a(R') = \Pr[\mathscr{E}' \mid \mathscr{F}']$. The law of conditional probability,

$$\Pr[\mathscr{E}] = \Pr[\mathscr{E} \mid \mathscr{F}](1 - \Pr[\bar{\mathscr{F}}]) + \Pr[\mathscr{E} \mid \bar{\mathscr{F}}] \Pr[\bar{\mathscr{F}}],$$

yields

$$\Pr[\mathscr{E} \mid \mathscr{F}] - \Pr[\bar{\mathscr{F}}] \leqq \Pr[\mathscr{E}] \leqq \Pr[\mathscr{E} \mid \mathscr{F}] + \Pr[\bar{\mathscr{F}}].$$

It follows from this and (5) (and from the analogous inequalities for the primed case) that

$$|a(P) - a(R)| \leqq 2\sqrt{\lambda m t} \quad \text{and} \quad |a(P') - a(R')| \leqq 2\sqrt{\lambda m t}. \tag{6}$$

We now define R to model P in the case that S produces a value uniformly in $(\lambda m, 1]$. The definition of R' in terms of P' is analogous. For each state i, let σ_i be the sum of p_{ij} over all j such that p_{ij} is small. For each i and j,

1) If p_{ij} is small, then $r_{ij} = 0$;
2) If p_{ij} is not small and j is not special for i, then $r_{ij} = p_{ij}/(1-\lambda m)$;
3) If j is special for i, then $r_{ij} = (p_{ij} + \sigma_i - \lambda m)/(1-\lambda m)$.

We must bound the closeness of r_{ij} to p_{ij} in the case 3) that j is special. Using $\sigma_i \leqq \lambda m$,

$$r_{ij} \leqq p_{ij}/(1-\lambda m).$$

Using $p_{ij} \geqq 1/m$,

$$r_{ij} \geqq (p_{ij}-\lambda m)/(1-\lambda m) \geqq p_{ij}(1-\lambda m^2)/(1-\lambda m).$$

Since P and P' are β-close mod λ, it follows that R and R' are $(\beta(1-\lambda m^2)^{-1})$-close. Using Lemma 4.1 and the bounds (6),

$$\begin{aligned} a(P') &\geqq a(R') - 2\sqrt{\lambda m t} \\ &\geqq \left(\frac{\beta}{1-\lambda m^2}\right)^{-2m} a(R) - 2\sqrt{\lambda m t} \\ &\geqq (1-2\lambda m^3)\beta^{-2m}(a(P)-2\sqrt{\lambda m t}) - 2\sqrt{\lambda m t} \\ &\geqq (1-2\lambda m^3)\beta^{-2m}a(P) - 4\sqrt{\lambda m t}. \end{aligned}$$ □

We can now prove the main lemma of this section.

LEMMA 4.3. *For every* $\varepsilon < \frac{1}{2}$ *there are positive constants* α_ε *and* η_ε *such that, if a* 2pfa M *having* c *states recognizes the language* L *within error probability* ε *and within expected time* $T(n)$, *then*

$$(7) \qquad (\alpha_\varepsilon c(\log T(n) + \log cn))^{c^2} \geqq N_L(n) \quad \textit{for all } n \geqq \eta_\varepsilon.$$

Proof. Let M be a 2pfa with c states which recognizes L within error probability $\varepsilon < \frac{1}{2}$ and within expected time $T(n)$. It is convenient to adopt a new starting convention by giving M a new initial state q_1. M is started on input $\not{c}x\not{c}$ in state q_1 with the head scanning the rightmost symbol of x; the machine then moves left across the input, remaining in state q_1, until the head reads the left $\not{c}$, at which point M enters its original initial state. Let $q_1, q_2, \cdots, q_{c+1}$ be the states of the modified M, where q_1 is the new initial state, q_c is the rejecting state, and q_{c+1} is the accepting state.

Note that $\sqrt{(1-\varepsilon)/2} > \frac{1}{2}$, since $\varepsilon < \frac{1}{2}$. Therefore, we can choose the constant η_ε so that

$$(1-2/n)\sqrt{(1-\varepsilon)/2} - 4\sqrt{1/n} > \tfrac{1}{2} \quad \text{for all } n \geqq \eta_\varepsilon.$$

Fix some $n \geqq \eta_\varepsilon$. Let W be a set of pairwise $\not\sim_n$ words with $|W| = N_L(n)$. For each $w \in W$, we define certain probabilities, called *word probabilities*, which capture the behavior of M on $\not{c}w$. A *starting condition* is an integer i with $1 \leqq i \leqq c-1$. A *stopping condition* is an integer j with $1 \leqq j \leqq c+1$. The starting condition i means "start M in state q_i on the word $\not{c}w$ with the head on the rightmost symbol of $\not{c}w$." We say that *M halts on $\not{c}w$ in state q_j* if either (i) $1 \leqq j \leqq c-1$ and the head moves off the right end of $\not{c}w$ at the same step as M enters state q_j, or (ii) $c \leqq j \leqq c+1$ and M enters state q_j before or at the step when the head moves off the right end of $\not{c}w$. If $j \leqq c-1$ or $j = c+1$, the stopping condition j means "M halts on $\not{c}w$ in state q_j." The stopping condition c means "either M halts on $\not{c}w$ in state q_c or M does not halt on $\not{c}w$." For each starting condition i and stopping condition j, let $q_{ij}(w)$ be the probability of the associated event. Let $Q(w)$ be the $(c-1)\times(c+1)$ matrix $Q(w) = \{q_{ij}(w)\}$.

Let

$$d = (c-1)(c+1), \quad m = 2c, \quad \lambda = (nm^3T(n))^{-1}.$$

Define an equivalence relation $\equiv$ on words in W as follows: $w \equiv w'$ if and only if, for all i and j, $q_{ij}(w) \leqq \lambda$ if and only if $q_{ij}(w') \leqq \lambda$. Let E be a largest equivalence class. Since there are at most 2^d equivalence classes, $|E| \geqq N_L(n)/2^d$.

For $w \in E$, let d' be the number of entries of $Q(w)$ which are strictly larger than λ. (The number and positions of these entries does not depend on the particular choice of w.) By projecting on these entries, map $Q(w)$ to an element $\mathbf{q}(w)$ of the space $[\lambda, 1]^{d'}$. Let $\log \mathbf{q}(w)$ be the componentwise log of $\mathbf{q}(w)$, so $\log \mathbf{q}(w) \in [\log \lambda, 0]^{d'}$. Let

$$\mu = \frac{\log(1-\varepsilon)+1}{4m}.$$

By dividing each coordinate interval $[\log \lambda, 0]$ into subintervals of length μ (with possibly one subinterval of length less than μ), we partition the space $[\log \lambda, 0]^{d'}$ into at most $\lceil(-\log \lambda)/\mu\rceil^d$ *cells*, each of size at most $\mu \times \mu \times \cdots \times \mu$.

We now show that

$$\lceil(-\log \lambda)/\mu\rceil^d \geqq N_L(n)/2^d. \tag{8}$$

This suffices to prove the lemma, since by substituting the values of λ, μ, and d into (8),

$$\left(\frac{16c(\log T(n) + \log(8nc^3))}{\log(1-\varepsilon)+1} + 2\right)^{(c-1)(c+1)} \geqq N_L(n).$$

So it is easy to see that the required α_ε exists.

Assuming that (8) does not hold, there must be two n-dissimilar words w, $w' \in E$ such that $\log \mathbf{q}(w)$ and $\log \mathbf{q}(w')$ belong to the same cell. Therefore, if q_{ij} and q'_{ij} are corresponding entries of $Q(w)$ and $Q(w')$, respectively, then either

1) $q_{ij} \leqq \lambda$ and $q'_{ij} \leqq \lambda$, or

2) $q_{ij} > \lambda$, $q'_{ij} > \lambda$, and $|\log q_{ij} - \log q'_{ij}| \leqq \mu$.

In the second case, q_{ij} and q'_{ij} are 2^μ-close.

Since $w \not\sim_n w'$, let v be such that $|wv| \leqq n$, $|w'v| \leqq n$, and $wv \in L$ if and only if $w'v \notin L$. By symmetry, assume that $wv \in L$ and $w'v \notin L$. We describe Markov chains P and P' which model the computation of M on wv and $w'v$, respectively. First define word probabilities for $v\not\epsilon$ similarly to the definition above for $\not\epsilon w$. The only difference is that starting conditions mean to start M in some state on the leftmost symbol of $v\not\epsilon$, and halting on $v\not\epsilon$ occurs when the head moves off the left end of $v\not\epsilon$ or when q_c or q_{c+1} is entered. For starting condition i with $1 \leqq i \leqq c-1$ and stopping condition j with $1 \leqq j \leqq c+1$, let $r_{ij}(v)$ be the probability of the associated event, and let $R(v)$ be the $(c-1)\times(c+1)$ matrix $\{r_{ij}(v)\}$.

The partial computations represented by $Q(w)$ and $R(v)$ are glued together by P in the obvious way. Formally, write $Q(w) = (Q_1(w)Q_2(w))$, $Q(w') = (Q_1(w')Q_2(w'))$, and $R(v) = (R_1(v)R_2(v))$, where $Q_1(w)$, $Q_1(w')$, and $R_1(v)$ are $(c-1)\times(c-1)$, and $Q_2(w)$, $Q_2(w')$, and $R_2(v)$ are $(c-1)\times 2$. Let I_2 be the 2×2 identity matrix. Then the $m \times m$ transition matrices P and P' are given by

$$P = \begin{pmatrix} 0 & Q_1(w) & Q_2(w) \\ R_1(v) & 0 & R_2(v) \\ 0 & 0 & I_2 \end{pmatrix}, \qquad P' = \begin{pmatrix} 0 & Q_1(w') & Q_2(w') \\ R_1(v) & 0 & R_2(v) \\ 0 & 0 & I_2 \end{pmatrix}.$$

Note that state 1 corresponds to starting M on the rightmost symbol of w (or w') in the new initial state q_1. State m corresponds to M accepting. Therefore, $a(P)$ (respectively, $a(P')$) is the probability that M accepts wv (respectively, $w'v$). In particular, $a(P) \geqq 1-\varepsilon$ since $wv \in L$. If $t = \max\{t(P), t(P')\}$, then $t \leqq T(n)$ since M runs in

expected time at most $T(n)$ on input wv or $w'v$. By choice of w and w', the Markov chains P and P' are 2^{μ}-close mod λ. Therefore, we can apply Lemma 4.2 to conclude

$$\begin{aligned}\Pr[M \text{ accepts } w'v] &\geqq (1-2\lambda m^3)2^{-2\mu m}(1-\varepsilon)-4\sqrt{\lambda m t}\\ &\geqq (1-2/n)2^{-(\log(1-\varepsilon)+1)/2}(1-\varepsilon)-4\sqrt{1/n}\\ &= (1-2/n)\sqrt{(1-\varepsilon)/2}-4\sqrt{1/n}\\ &> 1/2.\end{aligned}$$

The final inequality follows by the choice of η_ε and the assumption $n \geqq \eta_\varepsilon$. This is a contradiction since M recognizes L and $w'v \notin L$. □

The main result now follows immediately from Theorem 3.6 and Lemma 4.3.

THEOREM 4.4. *Let M be a* 2pfa *which recognizes a nonregular language within expected time $T(n)$. Then there is a constant $b>0$ such that $T(n) \geqq 2^{n^b}$ for infinitely many n.*

5. Turing machines with small space. Recall that if a deterministic or nondeterministic Turing machine has space bound $o(\log\log n)$, then the machine recognizes a regular language [7, Thm. 10.8]. The proof of the main result can easily be extended to prove the same result for probabilistic Turing machines with space bound $o(\log\log n)$, provided that the expected running time is not too large. The definition of a probabilistic Turing machine (ptm) is similar to the definition of a 2pfa, except that the machine has a fixed number of read/write worktapes. The ptm *M runs within space $S(n)$* if, on any input x with $|x| \leqq n$, at most $S(n)$ cells are used on any worktape. A ptm M can be viewed as a 2pfa with a growing number of states: On inputs of length at most n, the 2pfa has at most $c(n)=2^{dS(n)}$ states, where d is a constant which depends on M. If $S(n)=o(\log\log n)$ then, for any constant $a>0$, we have $c(n) \leqq (\log n)^a$ for all sufficiently large n. The proof of Lemma 4.3 remains valid for 2pfa's with a growing number $c(n)$ of states. Combining this observation with Theorem 3.6, the next result follows by a simple calculation which is left to the reader.

THEOREM 5.1. *Let M be a* ptm *which recognizes a nonregular language within space $o(\log\log n)$ and within expected time $T(n)$. Then for every $b<1$,*

$$\log\log T(n) \geqq (\log n)^b \quad \textit{for infinitely many } n.$$

In particular, $T(n)$ is not bounded above by any polynomial in n.

This result does not hold for larger space bounds. There is a deterministic Turing machine which recognizes a nonregular language within space $O(\log\log n)$ and time $O(n\log n)$ (see [7, Prob. 10.2]).

6. Number of states. A measure of the complexity of finite-state automata is the number of states. Some types of automata can be much more concise, that is, use many fewer states, than other types. Previous work has studied conciseness relationships among one-way and two-way deterministic and nondeterministic finite-state automata. A machine is *one-way* if the head can move only from left to right. For $j \in \{1,2\}$, a $j$dfa (respectively, $j$nfa) is a j-way deterministic (respectively, nondeterministic) finite state automaton. For example, Shepherdson [15] shows that any c-state 2dfa can be converted to an equivalent 1dfa having at most $(c+2)^{(c+1)}$ states. Conversely, Meyer and Fischer [11] describe a sequence of regular languages L_c, $c=1,2,\cdots$, such that L_c can be recognized by a 2dfa with $5c+5$ states, and any 1dfa recognizing L_c requires at least c^c states. In this section we make some observations about the conciseness of 2pfa's relative to other types of automata.

Methods of Freivalds [3] can be used to show that there is no recursive function f such that, for any c, if L is a regular language which can be recognized by a c-state 2pfa, then L can be recognized by some 1dfa with at most $f(c)$ states. In [3] it is stated that the emptiness problem for 2pfa's is undecidable. The proof (which is not given in [3] but which is easy to derive from techniques described there) actually shows the following. Any deterministic Turing machine Z can be effectively transformed to a 2pfa M such that (i) if Z halts on blank tape then M recognizes a finite nonempty language L, and (ii) if Z does not halt on blank tape then M recognizes the language $L=\varnothing$. In either case, L is regular. It is decidable, given a 2pfa M and an input word x, whether M accepts x with probability greater than $\frac{1}{2}$. Therefore, if there exists a recursive bound f as above, then the halting problem would be decidable.

With a sufficiently small bound on the expected running time of the 2pfa, a recursive bound f does exist. For simplicity, we focus on 2pfa's which run in polynomial expected time since this is a natural class. Generalizations to somewhat larger time bounds are possible and are left to the interested reader.

THEOREM 6.1. *For every $\varepsilon<\frac{1}{2}$, $a>0$ and $d>0$, there exists a constant $b>0$ such that, for any c, if L is recognized by a c-state* 2pfa *within error probability ε and within expected time an^d, then L is recognized by some* 1dfa *with at most c^{bc^2} states.*

Proof. Let L be as in the statement of the theorem. By Theorem 4.4, L is regular. Let k be the maximum of $N_L(n)$ over all n. Then some 1dfa with k states recognizes L (cf. the proof of Lemma 3.1). Let n' be such that $N_L(n'-1)<N_L(n')=k$. By Lemma 3.5, $n'\leqq 2k+k^2$. Let $n=2k+k^2$, so $N_L(n)=k$ since N_L is nondecreasing. If $n<\eta_\varepsilon$ where η_ε is the constant of Lemma 4.3, then $k<\eta_\varepsilon$, so we can choose b large enough to make the theorem true in this case. Assuming $n\geqq\eta_\varepsilon$ and noting that $n\leqq 3k^2$, the inequality (7) becomes

$$(\alpha_\varepsilon c(\log(a(3k^2)^d)+\log(3k^2c)))^{c^2}\geqq k. \tag{9}$$

A routine calculation shows that (9) implies $k\leqq c^{bc^2}$ for some b depending only on α_ε, a, and d. □

A large part of the exponential blow-up in Theorem 6.1 is necessary in the worst case since, as noted above, 2dfa's can be exponentially more concise than 1dfa's [11]. To study the effect of randomization on the conciseness of two-way automata, we must compare 2pfa's with 2dfa's or 2nfa's. The next result gives a lower bound on the improvement possible to the state bound c^{bc^2} in Theorem 6.1. Specifically, c^2 in the exponent cannot be replaced by a function of c which grows more slowly than $\sqrt{c/\log c}$, even if 1dfa is replaced by 2nfa.

The proof of this result uses a random walk technique of Greenberg and Weiss [5] which is also used in the next section. Let w be a nonempty word. A 2pfa M executes the procedure $RW(w)$ as follows. M begins with its head on the leftmost symbol of w. It then performs a random walk, moving the head right with probability $\frac{1}{2}$ or left with probability $\frac{1}{2}$ at each step. The procedure terminates when either (i) the head moves off the left end of w (outcome zero), or (ii) the head scans the rightmost symbol of w (outcome one). It is known (see, for example, [14, § 4.2]) that

1) The probability of outcome one is $1/|w|$, and

2) If $RW(w)$ is run repeatedly until it gives outcome one, then the expected total number of steps is $O(|w|^2)$.

THEOREM 6.2. (1) *For every ε with $0<\varepsilon<\frac{1}{2}$, there are constants a, $b>0$ such that, for every integer $m\geqq 4$ which is a power of two, there is a coin-tossing* 2pfa *M_m having at most $b\log^2 m/\log\log m$ states such that M_m recognizes $\{0^m\}$ within error probability ε and within expected time an^2.*

(2) *For every m, if the* 2nfa M *accepts the language* $\{0^m\}$, *then* M *has at least m states.*

Proof. (1) The proof combines the random walk technique with a method of Freivalds [3].

Let $m = 2^k$. The definition of M_m depends on two positive integer parameters B and D which are chosen below to make the error probability small. Although B and D depend on ε, they are independent of m. Let p_i denote the ith prime. Recall that $p_i = \Theta(i \log i)$. Let N be the smallest number such that $p_1 p_2 \cdots p_N \geqq Dm$, and note that $N = O(\log m / \log \log m)$.

The computation of M_m on input 0^n has three parts. The first two parts are deterministic.

First, check that $n \geqq \log m$, and reject if not.

Second, check that $n \equiv m \bmod p_i$ for all $1 \leqq i \leqq N$, and reject if not.

If 0^n passes the first two tests, then either $n = m$ or $n \geqq Dm$. The first test needs $O(\log m)$ states and takes time $O(n)$. The second test needs $O(\sum_{i=1}^{N} p_i)$ states. Since $p_i = O(\log m)$ for $i \leqq N$, the number of states is $O(\log^2 m / \log \log m)$. The second part takes time $O(Nn)$. Since $n \geqq \log m$ at this point, the time is $O(n^2)$.

The third part involves two probabilistic procedures $CF(k)$ and $RW(0^n)$. The random walk procedure $RW(0^n)$ is described above. To run $CF(k)$, M_m flips a fair coin k times; if all flips produce "heads," then the outcome is one, otherwise it is zero. Note that

$$\Pr[CF(k) = 1] = 2^{-k} = m^{-1},$$

$$\Pr[RW(0^n) = 1] = n^{-1}.$$

Let C_1 and C_2 be two counters which are initially zero. We can now describe the third test:

```
do until C1 + C2 = B
    begin
    if CF(k) = 1 then C1 ← C1 + 1;
    if RW(0^n) = 1 then C2 ← C2 + 1;
    end
if C1 > 0 and C2 > 0 then accept, else reject.
```

If $n = m$, then $\Pr[CF(k) = 1] = \Pr[RW(0^n) = 1]$. Therefore, for large B it is extremely unlikely (probability approaching zero as $B \to \infty$) that one counter will be incremented B times before the other counter is incremented once. Fix B sufficiently large. If $n \neq m$, then $n \geqq Dm$, so $\Pr[CF(k) = 1] \geqq D \cdot \Pr[RW(0^n) = 1]$. Therefore, for large D, it is extremely likely (probability approaching 1 as $D \to \infty$) that C_1 will reach value B before C_2 is incremented once.

It is easy to see that the third part needs $O(\log m)$ states and takes expected time $O(n^2)$.

(2) The proof is straightforward, and we only sketch the idea. Fix an m. Assume that the 2nfa M accepts the input 0^n if and only if $n = m$. Let the sequence $\{(s_t, h_t) | 1 \leqq t \leqq T\}$ be an accepting computation of M on input 0^m. The pair (s_t, h_t) gives the state s_t and head position h_t of M at time t. Head positions h_t satisfy $0 \leqq h_t \leqq m+1$; in particular, $h_t = 0$ (respectively, $h_t = m+1$) corresponds to reading the left (respectively, right) endmarker. A *right pass* (respectively, *left pass*) is a time interval $[y, z]$ such that $h_y = 0$ and $h_z = m+1$ (respectively, $h_y = m+1$ and $h_z = 0$) and $1 \leqq h_t \leqq m$ for all t with $y < t < z$.

Assume that M has fewer than m states. For each right pass $[y, z]$, we can find times i and j with $y < i < j < z$ such that $s_i = s_j$ and $h_i < h_j$. The situation for a left pass

is the same, except that $h_i > h_j$. Call $|h_i - h_j|$ the *period* of the pass (if a pass has more than one period, then choose one arbitrarily). It is not hard to see that if $n > m$ and n is congruent to m modulo the period of every pass, then M accepts 0^n, contradicting the requirement that M should accept 0^n if and only if $n = m$. □

7. Unbounded error probability. The proof of Lemma 4.3 uses the assumption that the error probability ε is bounded below $\frac{1}{2}$. Another model of probabilistic computation which has been considered in the literature (e.g., [4]) allows the error probability to approach $\frac{1}{2}$ as n increases. For a 2pfa M, define

$$L(M) = \{x \mid \Pr[M \text{ accepts } x] > \tfrac{1}{2}\}.$$

In the next result we observe that the main result, Theorem 4.4, does not hold in the unbounded error model.

THEOREM 7.1. *There is a coin-tossing* 2pfa M *which runs in expected time* $O(n^2)$ *such that* $L(M)$ *is not a regular language.*

Proof. We describe a 2pfa M such that

$$L(M) = \{0^a 1^b \mid 1 \leqq a \leqq b\}.$$

On input x, M first checks that $x = 0^a 1^b$ for some positive a and b. M then runs the following program until it halts:

1) If $RW(0^a) = 1$, then accept;
2) If $RW(1^b) = 1$, then reject, else go to 1.

Let

$$p_a = \Pr[RW(0^a) = 1] = 1/a,$$
$$p_b = \Pr[RW(1^b) = 1] = 1/b.$$

For $i \in \{1, 2\}$, let q_i be the probability that the program accepts when it is started at statement i. Since

$$q_1 = p_a + (1 - p_a)q_2, \qquad q_2 = (1 - p_b)q_1,$$

we have

$$\Pr[M \text{ accepts } 0^a 1^b] = q_1 = \frac{p_a}{p_a + p_b - p_a p_b}.$$

Substituting $p_a = 1/a$ and $p_b = 1/b$, it is easy to check that this probability is greater than $\frac{1}{2}$ if and only if $a \leqq b$. □

REFERENCES

[1] C. DWORK AND L. STOCKMEYER, *Interactive proof systems with finite state verifiers*, Report RJ 6262, IBM Research Division, San Jose, CA, May 1988.

[2] R. FREIVALDS, *Probabilistic machines can use less running time*, Information Processing '77, IFIP, North-Holland, Amsterdam, 1977, pp. 839-842.

[3] ———, *Probabilistic two-way machines*, in Proc. International Symposium on Mathematical Foundations of Computer Science, Lecture Notes in Computer Science, Vol. 118, Springer-Verlag, New York, 1981, pp. 33-45.

[4] J. GILL, *Computational complexity of probabilistic Turing machines*, SIAM J. Comput., 6 (1977), pp. 675-695.

[5] A. G. GREENBERG AND A. WEISS, *A lower bound for probabilistic algorithms for finite state machines*, J. Comput. System Sci., 33 (1986), pp. 88-105.

[6] G. H. HARDY, J. E. LITTLEWOOD, AND G. PÓLYA, *Inequalities*, 2nd ed., Cambridge University Press, Cambridge, U.K., 1959.

[7] J. E. HOPCROFT AND J. D. ULLMAN, *Formal Languages and Their Relation to Automata*, Addison-Wesley, Reading, MA, 1969.

[8] R. E. LADNER, R. J. LIPTON, AND L. J. STOCKMEYER, *Alternating pushdown and stack automata*, SIAM J. Comput., 13 (1984), pp. 135–155.

[9] F. T. LEIGHTON AND R. L. RIVEST, *The Markov chain tree theorem*, Report MIT/LCS/TM-249, Laboratory for Computer Science, Massachusetts Institute of Technology, Cambridge, MA, 1983.

[10] ———, *Estimating a probability using finite memory*, IEEE Trans. Inform. Theory, 32 (1986), pp. 733–742.

[11] A. R. MEYER AND M. J. FISCHER, *Economy of description by automata, grammars, and formal systems*, Conference Record 12th IEEE Symposium on Switching and Automata Theory, IEEE Computer Society, Washington, DC, 1971, pp. 188–191.

[12] M. O. RABIN, *Probabilistic automata*, Inform. and Control, 6 (1963), pp. 230–245.

[13] M. O. RABIN AND D. SCOTT, *Finite automata and their decision problems*, IBM J. Res. Develop., 3 (1959), pp. 114–125.

[14] E. SENETA, *Non-Negative Matrices and Markov Chains*, 2nd ed., Springer-Verlag, New York, 1981.

[15] J. C. SHEPHERDSON, *The reduction of two-way automata to one-way automata*, IBM J. Res. Develop., 3 (1959), pp. 198–200.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1024-1040, December 1990

004

THE SEARCHLIGHT SCHEDULING PROBLEM*

KAZUO SUGIHARA†, ICHIRO SUZUKI‡, AND MASAFUMI YAMASHITA§

Abstract. The problem of searching for a mobile robber in a simple polygon by a number of searchlights is considered. A searchlight is a stationary point which emits a single ray that cannot penetrate the boundary of the polygon. The direction of the ray can be changed continuously, and a point is detected by a searchlight at a given time if and only if it is on the ray. A robber is a point that can move continuously with unbounded speed. First, it is shown that the problem of obtaining a search schedule for an instance having at least one searchlight on the polygon boundary can be reduced to that for instances having no searchlight on the polygon boundary. The reduction is achieved by a recursive search strategy called the one-way sweep strategy. Then various sufficient conditions for the existence of a search schedule are presented by using the concept of a searchlight visibility graph. Finally, a simple necessary and sufficient condition for the existence of a search schedule for instances having exactly two searchlights in the interior is presented.

Key words. geometry, searchlight, visibility

AMS(MOS) subject classification. 68E99

1. Introduction.

We consider the problem of searching for a mobile robber in a simple polygon by a number of searchlights. A searchlight is a stationary point which emits a single ray. The ray cannot penetrate the boundary of the polygon, but its direction can be changed continuously. A point is detected at a given time if and only if it is on the ray of a searchlight. A robber is a point which can move continuously with unbounded speed. We refer to this problem as the *searchlight scheduling problem.* The objective is to decide whether there exists a search schedule for detecting a robber regardless of its movement, for a given instance. A possible application of the searchlight scheduling problem is security enforcement in industrial plants where searchlights or TV cameras are used to find an intruder.

In the searchlight scheduling problem, the locations of searchlights are given as part of a problem instance. Obviously, there exists a search schedule for an instance only if every point in the given polygon is visible from at least one searchlight. The problem of obtaining a set of locations of searchlights having this property is known as the art gallery problem [2]-[6].

First, we present a recursive search strategy called the one-way sweep strategy, and show that this strategy can be used to reduce the problem of obtaining a search schedule for an instance having at least one searchlight on the polygon boundary to that for instances having no searchlight on the polygon boundary. Next, we give a number of sufficient conditions for the existence of a search schedule by using the concept of a searchlight visibility graph which represents the visibility relations among searchlights. Finally, we consider the case in which no searchlight is located on the polygon boundary, and present a simple necessary and sufficient condition for the

* Received by the editors November 14, 1988; accepted for publication (in revised form) February 14, 1990. An earlier version of this paper was presented at the 26th Annual Allerton Conference on Communication, Control, and Computing, University of Illinois, Urbana, Illinois, September 28-30, 1988.

† Department of Information and Computer Sciences, University of Hawaii at Manoa, 2565 The Mall, Honolulu, Hawaii 96822.

‡ Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, P.O. Box 784, Milwaukee, Wisconsin 53201.

§ Department of Electrical Engineering, Faculty of Engineering, Hiroshima University, Shitami-Saijo, Higashi-Hiroshima 724, Japan.

existence of a search schedule for instances having exactly two searchlights in the interior.

It is not a goal of this paper to investigate the computational complexity of the problem. We also note that to our knowledge, the searchlight scheduling problem has not been addressed in the literature.

The problem is stated formally in § 2. The one-way sweep strategy is described in § 3. Searchlight visibility graphs and a number of sufficient conditions for the existence of a search schedule are discussed in § 4. Instances having two searchlights in the interior are considered in § 5. Concluding remarks are found in § 6.

2. Problem formulation. We denote by $b(R)$ the boundary of a two-dimensional region R. The term *simple polygon* is used to denote the union of a closed simple polygonal chain and its interior. For a simple polygon P and points $a, b \in b(P)$, $[a, b]_{b(P)}$ (or $(a, b)_{b(P)}$) denotes the closed (or open) continuous segment of $b(P)$ from a to b taken in the counterclockwise direction.

An instance of the searchlight scheduling problem is a pair $S = (P, L)$, where P is a simple polygon and L is a set of distinct points $l \in P$ called *searchlights*. A point x is said to be *visible* from a searchlight l if and only if $\overline{lx} \subseteq P$. Note that a searchlight does not block visibility from other searchlights. We denote by V_l the set of points visible from l.

DEFINITION 1. A *schedule* of a searchlight $l \in L$ is a continuous function $f_l : [0, T] \to \mathcal{R}$, where $[0, T]$ is an interval of real time and $\mathcal{R}$ is the set of real numbers. The *ray* of l at time $t \in [0, T]$ is the intersection of V_l and the semi-infinite ray with direction $f_l(t)$ emanating from l.[1] We say that l is *aimed* at a point $x \in P$ at time t if x is on the ray of l. A point $x \in P$ is said to be *illuminated* at time t if there exists a searchlight which is aimed at x.

DEFINITION 2. Two points in P are said to be *separable* at time $t \in [0, T]$ if every path between them within P contains an illuminated point; otherwise they are said to be *nonseparable*.

DEFINITION 3. Let $x \in P$ be any point.

(1) At time zero, x is *contaminated* if and only if x is not illuminated.

(2) At time $0 < t \leqq T$, x is *contaminated* if and only if there exists a point $y \in P$ such that (1) y is contaminated at some $0 \leqq t' < t$, (2) y is not illuminated at any time in the interval $[t', t]$, and (3) x and y are nonseparable at t.

A point which 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*.

It is easy to see that $x \in P$ is contaminated at $t \in [0, T]$ if and only if a robber who has not been detected in the interval $[0, t]$ can be located at x at t, where a robber is detected only when it is illuminated. Definition 3 is based on the assumption that a robber can move continuously with unbounded speed.

By definition, an illuminated point is clear, and a contaminated point remains contaminated until it is illuminated. The following lemma is immediate from the definition.

LEMMA 1. *At time $t \in [0, T]$, if two points x and $y \in P$ are nonseparable, then x is contaminated if and only if y is contaminated.*

By Lemma 1, a maximal contaminated region is a nonempty connected open region not containing any illuminated point, and hence it cannot consist only of points on the boundary of P. Therefore we have Lemma 2.

[1] The value of $f_l(t)$ is taken in radian. Directions are measured counterclockwise from the positive x-axis.

LEMMA 2. *Any maximal contaminated subregion of P contains a point in the interior of P.*

Our objective is to detect a robber in P regardless of the movement. Thus we have Definition 4.

DEFINITION 4. $F=\{f_l \mid f_l:[0, T]\to\mathcal{R}$ is a schedule of $l\in L\}$ is a *search schedule* for S if P is clear at T.

In the following, we describe a schedule of a searchlight l by using expressions such as "aim l at a point x" and "turn l clockwise," instead of specifying a function f_l explicitly.

Example 1. Consider the instance shown in Fig. 1. Searchlights l_1 and l_2 are aimed at point a at time zero. $(b, d)_{b(P)}$ is a maximal open segment of $b(P)$ not visible from l_1. If we turn l_1 counterclockwise from a to b without turning l_2, then the shaded region determined by segment $[a, b]_{b(P)}$ and the rays of l_1 and l_2 becomes clear. Since triangle bcd is still contaminated, the clear region becomes contaminated if l_1 is turned counterclockwise any further.

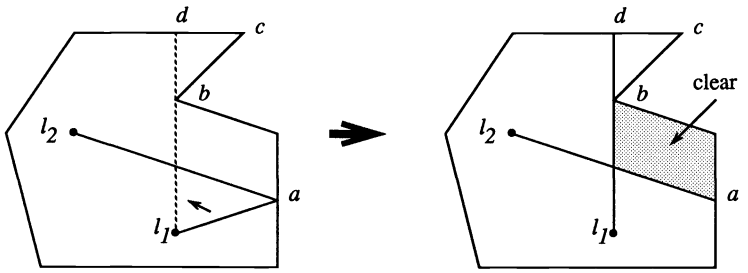


FIG. 1. *Illustration for Example* 1.

Example 2. The following is a search schedule for the instance shown in Fig. 2(a). Clear regions are shown shaded in Fig. 2.

(1) Aim l_2 at a.
(2) Aim l_3 at a and turn it counterclockwise until it is aimed at b (Fig. 2(b)).
(3) Aim l_1 at b and turn it counterclockwise until it is aimed at c (Fig. 2(c)).
(4) Turn l_3 counterclockwise until it is aimed at d (Fig. 2(d)).
(5) Aim l_1 at g.
(6) Turn l_2 clockwise until it is aimed at h (Fig. 2(e)).
(7) Turn l_1 counterclockwise until it is aimed at h (Fig. 2(f)).
(8) Turn l_1 clockwise until it is aimed at g (Fig. 2(g)).
(9) Turn l_3 counterclockwise until it is aimed at e (Fig. 2(h)).
(10) Aim l_2 at e and turn it counterclockwise until it is aimed at f.
(11) Turn l_3 counterclockwise until it is aimed at g (Fig. 2(i)).

An instance for which there exists no search schedule is given in Example 4 at the end of § 5.

Throughout this paper we assume that any given instance $S=(P, L)$ satisfies the following conditions (P1) and (P2), since obviously, otherwise there cannot exist any search schedule.

(P1) $P=\bigcup_{l\in L} V_l$. (Every point in P is visible from at least one searchlight.)

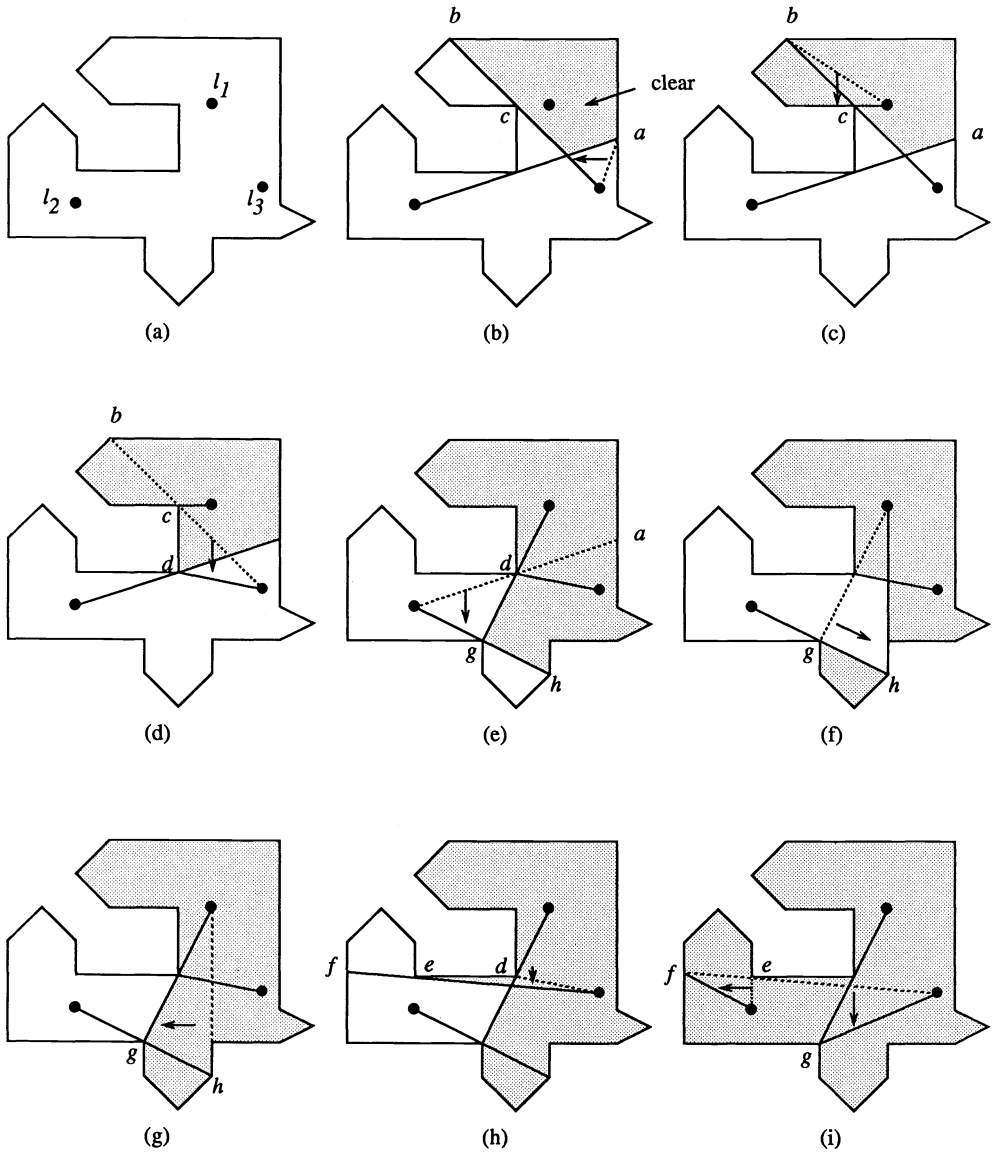


FIG. 2. *A search schedule for an instance of the searchlight scheduling problem.*

(P2) For each $l \in L$, either $l \in b(P)$ or $l \in V_{l'}$ for some $l' \in L - \{l\}$. (Every searchlight is either on the boundary of P or visible from another searchlight.)

3. One-way sweep strategy. In this section we show that the problem of obtaining a search schedule for an instance having at least one searchlight on the polygon boundary can be reduced to that for instances having no searchlight on the polygon boundary. The reduction is achieved by a recursive search strategy called the one-way sweep strategy.

It is convenient to describe the one-way sweep strategy as a method for clearing a subregion of P determined by the rays of searchlights. For this reason, we begin the discussion with the following definition.

DEFINITION 5. Let $S = (P, L)$ be an instance. *Semiconvex subpolygons* of P *supported* by a set of searchlights at a given time are defined recursively as follows.

(1) P is a semiconvex subpolygon of P supported by $\emptyset$ at any time $t \geqq 0$.

(2) Let $R \subseteq P$ be a semiconvex subpolygon of P supported by $K \subset L$ at time $t \geqq 0$. For an arbitrary searchlight $l \in L - K$ and an arbitrary maximal open segment $(a, b)_{b(P)}$ of $b(P)$ not visible from l, let Q be the closed simple region whose boundary is $[a, b]_{b(P)} \cup \overline{ba}$. If (1) $R \cap Q \neq \emptyset$ and (2) l is aimed at a and b at t, then $R \cap Q$ is a semiconvex subpolygon of P supported by $K \cup \{l\}$ at t.

In Fig. 3, the boundary of a semiconvex subpolygon R supported by $K = \{l_1, l_2\}$ is shown in thick lines.

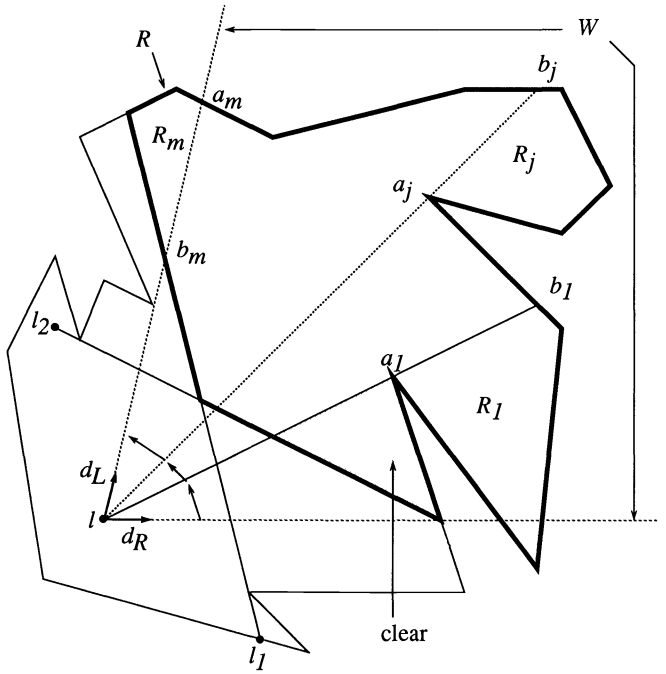


FIG. 3. *The one-way sweep strategy* OWSS (R, K, l), *where* $K = \{l_1, l_2\}$.

If R is supported by K at time t, then (1) it is "enclosed" by a segment of $b(P)$ and the rays of (some of) the searchlights in K, and (2) the interior of R is not visible from any searchlight in K. In the following, the qualifier "at time t" may be omitted when it is understood from the context. The term "semiconvex" is due to the following fact which is straightforward from definition: any reflex vertex of R is a vertex of P.

Let $S = (P, L)$ be an instance. Let R be a semiconvex subpolygon of P supported by $K \subset L$. Suppose that there exists a searchlight $l \in L - K$ such that

(1) $l \notin R - b(R)$ (l is either on the boundary of R or external to R), and

(2) $(R - b(R)) \cap V_l \neq \emptyset$ (at least one point in the interior of R is visible from l).

Let W be the smallest wedge with apex l such that $R \cap V_l \subseteq W$. Let d_R and d_L be the bounding semi-infinite rays of W where the interior of W lies to the left of d_R and to the right of d_L. Let $(a_j, b_j)_{b(R)} \subseteq b(R) - V_l$, $1 \leqq j \leqq m$, be the maximal open segments of $b(R)$ not visible from l, where the line segments $\overline{a_1 b_1}$, $\overline{a_2 b_2}$, $\cdots$, $\overline{a_m b_m}$ appear in

counterclockwise order within W when viewed from l. Let R_j be the closed simple region whose boundary is $[a_j, b_j]_{b(R)} \cup \overline{b_j a_j}$ (see Fig. 3). Then the *one-way sweep strategy* OWSS (R, K, l) for R (with respect to K and l) is the following.

OWSS (R, K, l)
1. Aim l in the direction of d_R.
2. **for** $j = 1$ **to** m **do**
   2.1. Turn l counterclockwise until it is aimed at a_j and b_j.
   2.2. If there exists a searchlight $l' \in L - (K \cup \{l\})$ such that $l' \notin R_j - b(R_j)$ (l' is either on the boundary of R_j or external to R_j) and $(R_j - b(R_j)) \cap V_{l'} \neq \emptyset$ (at least one point in the interior of R_j is visible from l'), then execute OWSS $(R_j, K \cup \{l\}, l')$. Otherwise, if there exists a search schedule for the instance $S_{R_j} = (R_j, L \cap R_j)$, then execute it; otherwise output **failure** and halt.
3. Turn l counterclockwise until it is aimed in the direction of d_L.

In OWSS (R, K, l), we clear R by sweeping it by l in one direction, in such a way that every region R_j not visible from l is cleared in step 2.2 (if possible) without turning any searchlight in $K \cup \{l\}$. Since R is supported by K, it is easy to see that if each R_j can be cleared without turning any searchlight in $K \cup \{l\}$, then R becomes clear when step 3 is completed.

In step 2.2, to clear R_j we apply the one-way sweep strategy recursively if there exists a searchlight $l' \in L - (K \cup \{l\})$ which is not in the interior of R_j and from which at least one point in the interior of R_j is visible. Note that the idea of applying the strategy to R_j is valid, since R_j is a semiconvex subpolygon of P supported by $K \cup \{l\}$ when l is aimed at a_j and b_j. If there exists no such l', then the interior of R_j is visible only from the searchlights in the interior of R_j (and hence there exists no searchlight on the boundary of R_j, since at least one point in the interior of R_j would be visible from any searchlight on the boundary of R_j). In this case we regard $S_{R_j} = (R_j, L \cap R_j)$ as a separate instance and clear R_j by executing a search schedule for S_{R_j}, if such a search schedule exists. If there exists no search schedule for S_{R_j}, then the strategy outputs **failure** and halts.

THEOREM 1. *Let $S = (P, L)$ be an instance. Let R be a semi-convex subpolygon of P supported by $K \subset L$. Suppose that there exists a searchlight $l \in L - K$ such that $l \notin R - b(R)$ (l is either on the boundary of R or external to R) and $(R - b(R)) \cap V_l \neq \emptyset$ (at least one point in the interior of R is visible from l). Then R can be cleared without turning any searchlight in K if and only if there exists a search schedule for the instance $S_Q = (Q, L \cap Q)$ for every semiconvex subpolygon Q of R found during the execution of* OWSS (R, K, l) *to which the strategy cannot be applied recursively.*

Proof. (If) Execute OWSS (R, K, l). As is discussed above, R becomes clear when the execution terminates, since (1) R is supported by K and (2) every semiconvex subpolygon Q of R found during the execution of OWSS (R, K, l) can be cleared either by a recursive application of the one-way sweep strategy or the execution of a search schedule for the instance $S_Q = (Q, L \cap Q)$.

(Only if) Let Q be a semiconvex subpolygon Q of R found during the execution of OWSS (R, K, l) to which the strategy cannot be applied recursively. Let $F = \{f_l : [0, T] \to \mathcal{R} \mid l \in L - K\}$ be a collection of schedules which clears R without turning any searchlight in K starting from the state in which R is supported by K. Suppose that there exists no search schedule for $S_Q = (Q, L \cap Q)$. Then Q is contaminated when the execution of $F_Q = \{f_l \in F \mid l \in Q\}$ terminates at T, and hence by Lemma 2 there exists a contaminated point x in the interior of Q. Here, since the interior of Q is visible

only from the searchlights in the interior of Q, for any $0 \leqq t \leqq T$, a point in the interior of Q is illuminated at t during the execution of F_Q if and only if it is illuminated at t during the execution of F. This, together with Lemma 2, implies that x is contaminated when the execution of F terminates at T. This contradicts the assumption that F clears R. □

Let $S = (P, L)$ be an instance having at least one searchlight on the boundary of P, and let $l \in L \cap b(P)$ be an arbitrary searchlight on the boundary of P. Since P is a semiconvex subpolygon of P supported by $\varnothing$ and at least one point in the interior of P is visible from l, we can execute OWSS $(P, \varnothing, l)$. Then by Theorem 1, there exists a search schedule for S if and only if there exists a search schedule for the instance $S_Q = (Q, L \cap Q)$ for every semiconvex subpolygon Q of P found during the execution of OWSS $(P, \varnothing, l)$ to which the strategy cannot be applied recursively. Since there exists no searchlight on the boundary of such Q, the problem of finding a search schedule for an instance having at least one searchlight on the polygon boundary has been reduced to that for instances having no searchlight on the polygon boundary.

Example 3. Consider the instance $S = (P, \{l_1, l_2, l_3, l_4\})$ shown in Fig. 4. It is easy to see that the one-way sweep strategy can be recursively applied to every semiconvex subpolygon of P found during the execution of OWSS $(P, \varnothing, l_1)$, and hence by Theorem 1 there exists a search schedule for S.

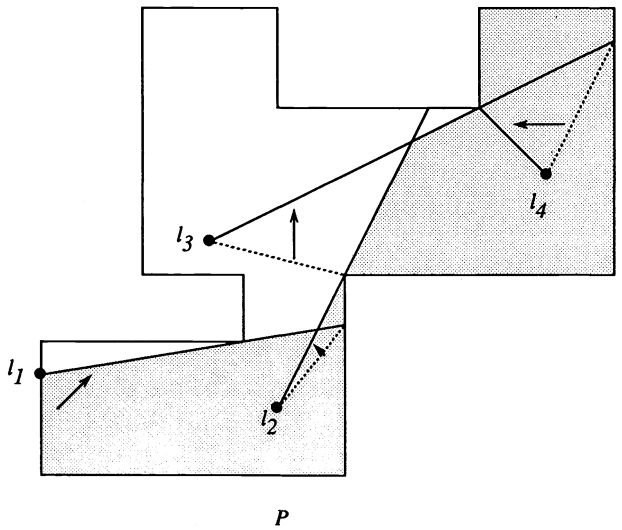


FIG. 4. *An instance having a search schedule.*

4. Searchlight visibility graphs. In this section we present a number of simple sufficient conditions for the existence of a search schedule. The conditions are stated by using the concept of a searchlight visibility graph introduced below.

DEFINITION 6. Let $S = (P, L)$ be an instance. The *searchlight visibility graph* of S is an undirected graph SVG $(S) = (L, E)$ with vertex set L and edge set E such that for any l and $l' \in L$, $(l, l') \in E$ if and only if $l \neq l'$ and $l \in V_{l'}$.

THEOREM 2. *Let $S = (P, L)$ be an instance. There exists a search schedule for S if for every connected component $G_i = (L_i, E_i)$ of* SVG (S), *there exists at least one searchlight $l \in L_i$ such that $l \in b(P)$.*

Proof. Suppose that we execute OWSS $(P, \varnothing, l)$, where $l \in L \cap b(P)$ is an arbitrary searchlight on the boundary of P. By Theorem 1, it suffices to show that the one-way sweep strategy can be applied recursively to any semiconvex subpolygon Q of P found during the execution of OWSS $(P, \varnothing, l)$. Suppose that the strategy cannot be applied to some Q. Consider the instance $S_Q = (Q, L \cap Q)$. Note that the interior of Q is visible only from the searchlights in the interior of Q and there exists no searchlight on the boundary of Q. This observation, together with condition (P1), implies that (1) there exists at least one searchlight in the interior of Q, (2) any connected component of SVG (S_Q) is a connected component of SVG (S), and (3) $L_i \cap b(P) = \varnothing$ for any connected component $G_i = (L_i, E_i)$ of SVG (S_Q). This contradicts the assumption. □

LEMMA 3. *Let $S = (P, L)$ be an instance. For an arbitrary searchlight $l \in L$, let $(a, b)_{b(P)} \subseteq b(P) - V_l$ be a maximal open segment of $b(P)$ not visible from l, and let R be the closed simple region whose boundary is $[a, b]_{b(P)} \cup \overline{ba}$. If* SVG (S) *is connected, then R can be cleared while l is kept aimed at a and b.*

Proof. Aim l at a and b (Fig. 5). Then R is a semiconvex subpolygon of P supported by $\{l\}$. By condition (P1) and the connectedness of SVG (S), there exists a searchlight l' such that $l' \notin R - b(R)$ and $(R - b(R)) \cap V_{l'} \neq \varnothing$. Thus we can execute OWSS $(R, \{l\}, l')$. By Theorem 1, it suffices to show that the one-way sweep strategy can be applied recursively to any semiconvex subpolygon Q of R found during the execution of OWSS $(R, \{l\}, l')$. Suppose that the strategy cannot be applied recursively to some Q. By condition (P1) and the fact that the interior of Q is visible only from the searchlights in the interior of Q, there exists at least one searchlight in the interior of Q. But then the searchlights in the interior of Q are not visible from any searchlight outside of Q, and thus SVG (S) cannot be connected. □

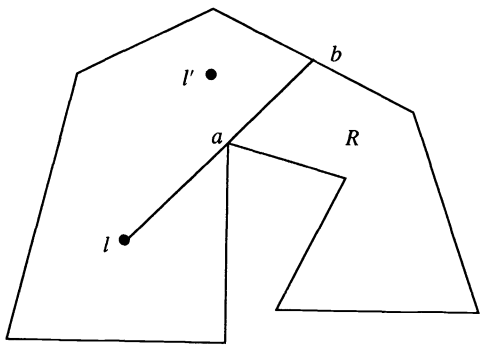


FIG. 5. *Illustration for Lemma 3; l is aimed at a and b.*

THEOREM 3. *Let $S = (P, L)$ be an instance. If* SVG (S) *is connected, then there exists a search schedule for the instance $S' = (P, L \cup \{l'\})$, where $l' \in P$ is an arbitrary searchlight not in L.*

Proof. By condition (P1), l' is visible from some searchlight $l \in L$. Let p be the first intersection of $b(P)$ and the ray emanating from l in the direction from l' to l (Fig. 6). Aim l and l' at p, and then turn l counterclockwise through a rotation of 2π. During this rotation, whenever l is aimed at points a and $b \in b(P)$ such that $(a, b)_{b(P)} \subseteq b(P) - V_l$ is a maximal open segment of $b(P)$ not visible from l, clear the closed region R whose boundary is $[a, b]_{b(P)} \cup \overline{ba}$ without turning l. This is possible by Lemma 3, since SVG (S) is connected and R is a semiconvex subpolygon of P supported by $\{l\}$ when l is aimed at a and b. Since l' need not be turned while R is being cleared, P becomes clear when the rotation of l is completed. □

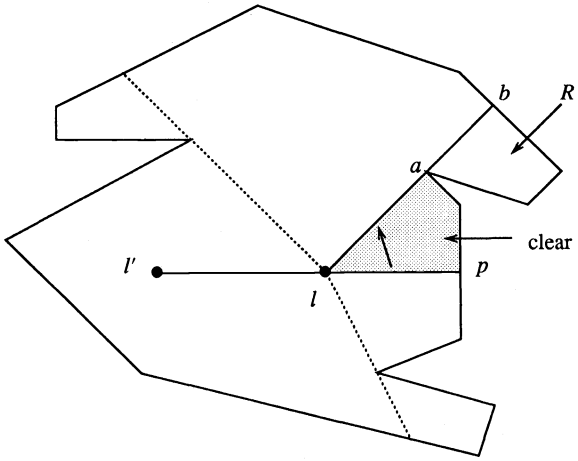


FIG. 6. *An additional searchlight l'.*

THEOREM 4. *Let $S=(P, L)$ be an instance. If* SVG (S) *is connected and there exist two searchlights l and $l' \in L$ such that $V_l \cap V_{l'} = \emptyset$, then there exists a search schedule for P.*

Proof. Let $(a, b)_{b(P)} \subseteq b(P) - V_l$ be the maximal open segment of $b(P)$ not visible from l such that $l' \in R$, where R is the closed simple region whose boundary is $[a, b]_{b(P)} \cup \overline{ba}$. Similarly, let $(a', b')_{b(P)} \subseteq b(P) - V_{l'}$ be the maximal open segment of $b(P)$ not visible from l' such that $l \in R'$, where R' is the closed simple region whose boundary is $[a', b']_{b(P)} \cup \overline{b'a'}$ (Fig. 7). Since SVG (S) is connected, by Lemma 3 we can aim l at a and b and then clear R without turning l. At this state $P - R'$ is clear, since $V_l \cap V_{l'} = \emptyset$. Next, we aim l' at a' and b' and clear R' without turning l'. Again, this is possible by Lemma 3. Then P becomes clear. □

5. Instances having two interior searchlights. In this section we present a simple necessary and sufficient condition for the existence of a search schedule for instances having exactly two searchlights in the interior.

THEOREM 5. *Let $S=(P, \{l_1, l_2\})$ be an instance such that $l_1, l_2 \notin b(P)$. Let p (or q) be the first intersection of the boundary of P and the extension of $\overline{l_1 l_2}$ in the direction from l_2 to l_1 (or from l_1 to l_2). Let $W_u = [p, q]_{b(P)}$ and $W_l = [q, p]_{b(P)}$. There exists a search schedule for P if and only if one of the following conditions holds.*

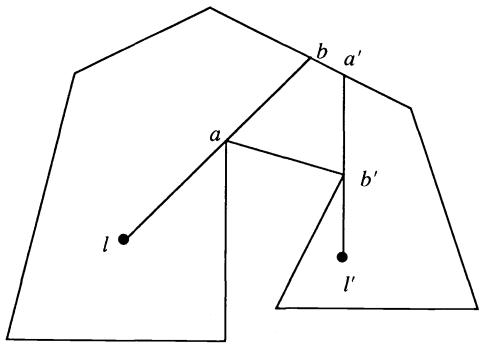


FIG. 7. *Illustration for Theorem* 4.

(1) *There exist points* $c_u \in W_u$ *and* $c_l \in W_l$ *such that* $[c_u, c_l]_{b(P)} \subseteq V_{l_1}$ *and* $[c_l, c_u]_{b(P)} \subseteq V_{l_2}$.

(2) $\overline{l_1 l_2} \cap W_u \neq \emptyset$ *and* $\overline{l_1 l_2} \cap W_l \neq \emptyset$.

(3) $\overline{l_1 l_2} \cap W_u \neq \emptyset$ *and either* $W_l \subseteq V_{l_1}$ *or* $W_l \subseteq V_{l_2}$.

(4) $\overline{l_1 l_2} \cap W_l \neq \emptyset$ *and either* $W_u \subseteq V_{l_1}$ *or* $W_u \subseteq V_{l_2}$.

Note that S is assumed to satisfy conditions (P1) and (P2) given at the end of § 2. Since $\overline{l_1 l_2} \cap b(P) = \emptyset$ holds if there exist points $c_u \in W_u$ and $c_l \in W_l$ such that $[c_u, c_l]_{b(P)} \subseteq V_{l_1}$ and $[c_l, c_u]_{b(P)} \subseteq V_{l_2}$, Theorem 5 follows from Lemmas 4 and 5 given below.

LEMMA 4. *If* $\overline{l_1 l_2} \cap b(P) = \emptyset$, *then there exists a search schedule for* P *if and only if there exist points* $c_u \in W_u$ *and* $c_l \in W_l$ *such that* $[c_u, c_l]_{b(P)} \subseteq V_{l_1}$ *and* $[c_l, c_u]_{b(P)} \subseteq V_{l_2}$.

Proof. (If) The following is a search schedule for P (Fig. 8).

(1) Aim l_1 at c_u.

(2) Aim l_2 at c_u.

(3) Turn l_1 counterclockwise until it is aimed at q.

(4) Turn l_2 clockwise until it is aimed at p.

(5) Turn l_1 counterclockwise until it is aimed at c_l.

(6) Turn l_2 clockwise until it is aimed at c_l.

(Only if) Assume that such $c_u \in W_u$ and $c_l \in W_l$ do not exist. We consider the case in which there exist maximal open segments $(a_1, b_1)_{b(P)} \subseteq W_u - V_{l_2}$ and $(a_2, b_2)_{b(P)} \subseteq W_u - V_{l_1}$ not visible from l_2 and l_1, respectively, such that a_1, b_1, a_2, and b_2 appear in counterclockwise order in W_u (Fig. 9). The argument for the case in which there exist similar open segments in W_l is basically the same. By $\overline{l_1 l_2} \cap b(P) = \emptyset$ and condition (P1), we have $a_1, b_1, a_2, b_2 \notin \overline{pq}$. We may assume that a_1, b_1, a_2, and b_2 have been chosen so that $[p, a_1]_{b(P)} \subseteq V_{l_2}$ and $[b_2, q]_{b(P)} \subseteq V_{l_1}$. For $i = 1, 2$, let R_i be the closed simple region whose boundary is $[a_i, b_i]_{b(P)} \cup \overline{b_i a_i}$. Let R_0 be the closed region whose boundary is $W_l \cup \overline{pq}$.

Before we proceed, we prove the following proposition.

PROPOSITION 1. *In any search schedule for* P, *if* R_1 *is changed from contaminated to clear at time* t, *then there exists some* $\delta > 0$ *such that in the interval* $[t - \delta, t)$, l_1 *is aimed at a point in* $(a_1, b_1)_{b(P)}$ *and* l_2 *is aimed at a point in* $[p, a_1]_{b(P)}$.

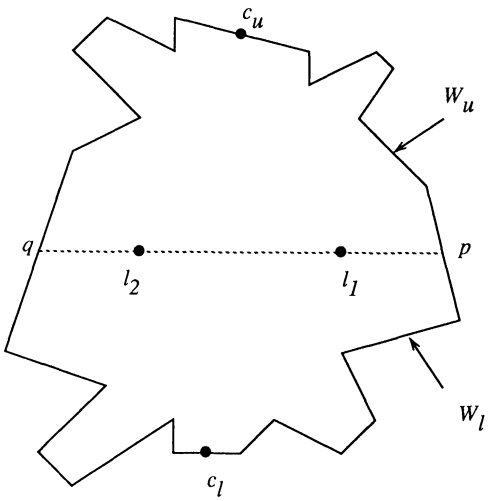


FIG. 8. *Points* c_u *and* c_l.

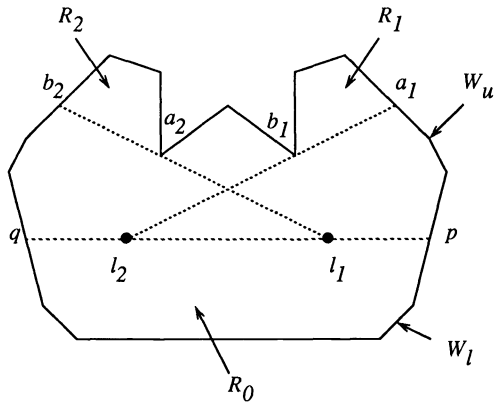


FIG. 9. *Illustration for the proof of Lemma* 4.

Proof. Let $\delta > 0$ be any value such that R_1 is contaminated in $[t-\delta, t)$. Suppose that in $[t-\delta, t)$, either l_1 is not aimed at any point in $(a_1, b_1)_{b(P)}$ or l_2 is not aimed at any point in $[p, a_1]_{b(P)}$ (Fig. 10). At any time in $[t-\delta, t)$, since R_1 is contaminated and any two points in R_1 which are not illuminated are nonseparable, by Lemma 1 any point in R_1 which is not illuminated is contaminated. Then it is impossible to change R_1 from contaminated to clear at t, since contaminated points remain contaminated until they are illuminated. □

The proof of the following proposition is basically the same as that of Proposition 1 and is thus omitted.

PROPOSITION 2. *In any search schedule for P, if R_2 is changed from contaminated to clear at time t, then there exists some $\delta > 0$ such that in the interval $[t-\delta, t)$, l_2 is aimed at a point in $(a_2, b_2)_{b(P)}$ and l_1 is aimed at a point in $[b_2, q]_{b(P)}$.*

We return to the proof of Lemma 4. Assume that there exists a search schedule for P. Let F be a search schedule in which the total number of times R_1 and R_2 are changed from contaminated to clear is smallest among all search schedules. Suppose that during the execution of F, R_1 is changed from contaminated to clear at t_1 and R_1 remains clear after t_1. Since R_1 and R_2 cannot be changed from contaminated to clear simultaneously by Propositions 1 and 2, without loss of generality assume that R_2 is contaminated at t_1 or at some time after t_1. Let $t_2 > t_1$ be the first time after t_1 at which R_2 is changed from contaminated to clear.

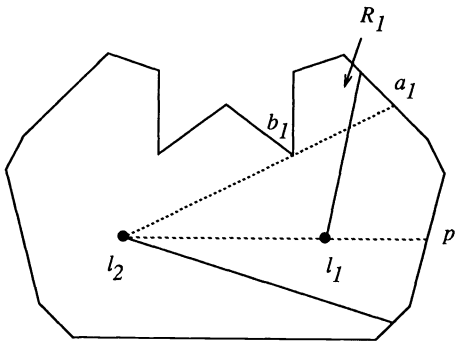


FIG. 10. *Illustration for the proof of Proposition* 1; *any two points in R_1 which are not illuminated are nonseparable.*

First, we show that both R_0 and R_2 are contaminated at t_1. Let $\delta_1 > 0$ be a value satisfying the conditions of Proposition 1 with respect to t_1, that is, l_1 is aimed at a point in $(a_1, b_1)_{b(P)}$ and l_2 is aimed at a point in $[p, a_1]_{b(P)}$ in $[t_1 - \delta_1, t_1)$. Then by the assumption that $\overline{l_1 l_2} \cap W_l = \emptyset$, in $[t_1 - \delta_1, t_1)$ any two points in $R_0 \cup R_2$ which are not illuminated are nonseparable, and hence by Lemma 1 either R_0 and R_2 are both clear or both contaminated. Suppose that R_0 and R_2 are clear in $[t_1 - \delta_1, t_1)$, and hence by Lemma 1 the points in $R_0 \cup R_2$ are separable from any contaminated point. Since $[p, a_1]_{b(P)} \subseteq V_{l_2}$ and $\overline{l_1 l_2} \cap W_l = \emptyset$, there are only two possibilities at any time in $[t_1 - \delta_1, t_1)$.

Case 1. l_2 is not aimed at a_1, and the region determined by some segment of $[p, b_1)_{b(P)}$ and the rays of l_1 and l_2 is the only contaminated region (Fig. 11).

Case 2. l_2 is aimed at a_1, and some of the regions determined by some segments of $[a_1, a_2]_{b(P)}$ and the rays of l_1 and l_2 are the only contaminated regions (Fig. 12). (Without the assumption that $[p, a_1]_{b(P)} \subseteq V_{l_2}$, there may exist a contaminated region determined by the ray of l_2 and some segments of $[p, a_1]_{b(P)}$. Also, if $\overline{l_1 l_2} \cap W_l \neq \emptyset$, then there may exist a contaminated region determined by the ray of l_2 and some segments of W_l.) In Case 1, P can be cleared by turning l_1 and l_2 to a_1 clockwise and counterclockwise, respectively. In Case 2, the contaminated regions are visible from l_1 by condition (P1), and thus P can be cleared without changing any of R_1 and R_2 from clear to contaminated after t_1. In either case, there exists a search schedule for S in which the number of times R_1 and R_2 are changed from contaminated to clear is smaller than that in F. Since this contradicts the assumption on F, it cannot be the case that R_0 and R_2 are clear in $[t_1 - \delta_1, t_1)$. Thus both R_0 and R_2 are contaminated in $[t_1 - \delta_1, t_1)$. Then, since by Proposition 1 it is impossible to change either of R_0 and

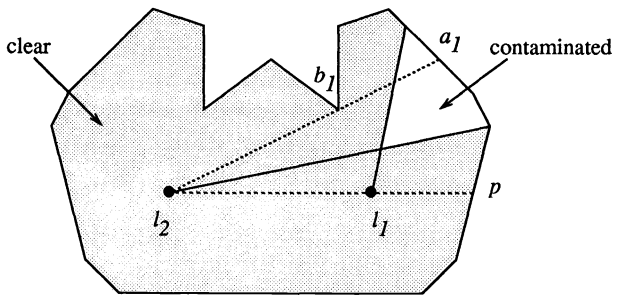


FIG. 11. *Case* 1 *in* $[t_1 - \delta_1, t_1)$ *in the proof of Lemma* 4.

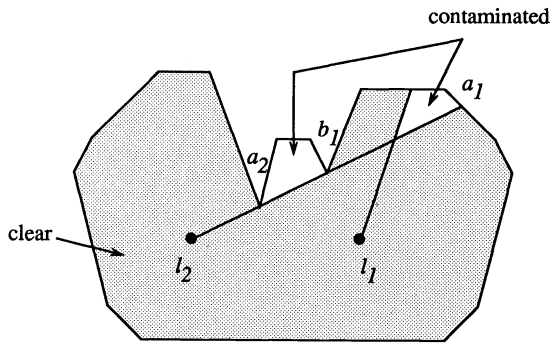


FIG. 12. *Case* 2 *in* $[t_1 - \delta_1, t_1)$ *in the proof of Lemma* 4.

R_2 from contaminated to clear at t_1, both R_0 and R_2 are contaminated at t_1. Also, note that by the argument given above, q is contaminated at t_1 since $a_2, b_2 \notin \overline{pq}$.

Let $\delta_2 > 0$ be a value satisfying the conditions of Proposition 2 with respect to t_2, that is, l_2 is aimed at a point in $(a_2, b_2)_{b(P)}$ and l_1 is aimed at a point in $[b_2, q]_{b(P)}$ in $[t_2 - \delta_2, t_2)$. Then by the assumption that $\overline{l_1 l_2} \cap W_l = \emptyset$, in $[t_2 - \delta_2, t_2)$ any two points in $R_0 \cup R_1$ which are not illuminated are nonseparable. Thus by Lemma 1 and the assumption that R_1 remains clear after t_1, R_0 is clear in $[t_2 - \delta_2, t_2)$.

In summary, we have found that R_1 is clear in $[t_1, t_2]$, R_2 is contaminated in $[t_1, t_2)$, R_0 is contaminated at t_1, and R_0 is changed from contaminated to clear in $[t_1, t_2)$. In the following we show that at least one of p and q is contaminated at any time in $[t_1, t_2)$, and hence R_0 cannot become clear in $[t_1, t_2)$.

Since in $[t_1, t_2)$ R_1 is clear and R_2 is contaminated, by Lemma 1 the points in R_1 should be separable from any contaminated point in R_2. Thus we have Proposition 3.

PROPOSITION 3. *In the interval* $[t_1, t_2)$,

(1) *Whenever* l_1 *is aimed at* p, l_2 *is aimed at* a_1 *and* b_1 (*Fig.* 13), *and*

(2) *Whenever,* l_2 *is aimed at* q, l_1 *is aimed at* a_2 *and* b_2.

Also, by Lemma 1, $a_1, b_1, a_2, b_2 \notin \overline{pq}$ and the condition on R_1 and R_2, we have Proposition 4.

PROPOSITION 4. *In the interval* $[t_1, t_2)$,

(1) *Whenever* l_1 *is aimed at* q, l_2 *is aimed at a point in* $[b_1, a_2]$ (*Fig.* 14), *and*

(2) *Whenever* l_2 *is aimed at* p, l_1 *is aimed at a point in* $[b_1, a_2]$.

Furthermore, since $a_1, b_1, a_2, b_2 \notin \overline{pq}$, we have Proposition 5.

PROPOSITION 5. *At any time, if neither* p *nor* q *is illuminated, then* p *and* q *are nonseparable.*

By $a_1, b_1, a_2, b_2 \notin \overline{pq}$ and Propositions 3 and 4, (1) at most one of p and q is illuminated at any time in $[t_1, t_2)$, and (2) if p and q (or q and p) are illuminated at

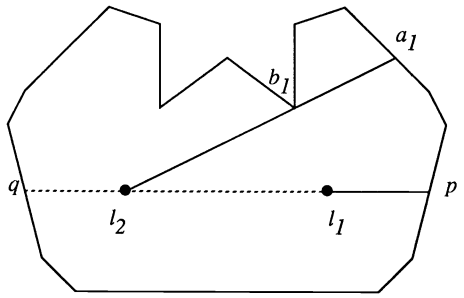


FIG. 13. *Illustration for Proposition* 3; l_1 *is aimed at* p.

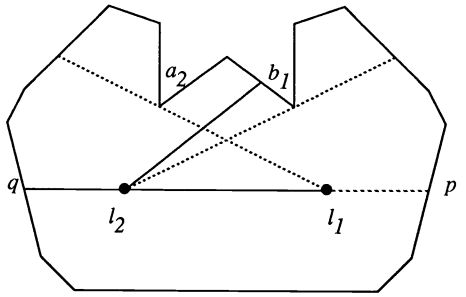


FIG. 14. *Illustration for Proposition* 4; l_1 *is aimed at* q.

s_1 and s_2 for some $t_1 \leqq s_1 < s_2 < t_2$, respectively, then there exists some $s_1 < t < s_2$ such that neither p nor q is illuminated at t. This observation, together with Proposition 5, Lemma 1, and the fact that q is contaminated at t_1, implies that p and q cannot be clear simultaneously in $[t_1, t_2)$. Thus R_0 cannot be clear in $[t_1, t_2)$. This is a contradiction. □

LEMMA 5. *If $\overline{l_1 l_2} \cap b(P) \neq \emptyset$, then there exists a search schedule for P if and only if one of the following conditions holds*:

(1) $\overline{l_1 l_2} \cap W_u \neq \emptyset$ *and* $\overline{l_1 l_2} \cap W_l \neq \emptyset$.

(2) $\overline{l_1 l_2} \cap W_u \neq \emptyset$ *and either* $W_l \subseteq V_{l_1}$ *or* $W_l \subseteq V_{l_2}$.

(3) $\overline{l_1 l_2} \cap W_l \neq \emptyset$ *and either* $W_u \subseteq V_{l_1}$ *or* $W_u \subseteq V_{l_2}$.

Proof. (If) Note that $\overline{l_1 l_2} \subseteq P$ by condition (P2). The following is a search schedule for P if $\overline{l_1 l_2} \cap W_u \neq \emptyset$ and $\overline{l_1 l_2} \cap W_l \neq \emptyset$ (Fig. 15).

(1) Aim l_1 at q.

(2) Aim l_2 at p.

(3) Turn l_1 counterclockwise through a rotation of 2π.

(4) Turn l_2 counterclockwise through a rotation of 2π.

If $\overline{l_1 l_2} \cap W_u \neq \emptyset$ and $W_l \subseteq V_{l_1}$, then P can be cleared by the following (Fig. 16).

(1) Aim l_1 at q.

(2) Aim l_2 at p.

(3) Turn l_1 clockwise through a rotation of 2π.

(4) Turn l_2 counterclockwise through a rotation of π.

Search schedules for other cases are similar and are thus omitted.

(Only if) Since the argument is similar to that in the (only if) part of Lemma 4, we only give an outline. Consider the case in which $\overline{l_1 l_2} \cap W_u \neq \emptyset$, $\overline{l_1 l_2} \cap W_l = \emptyset$, and

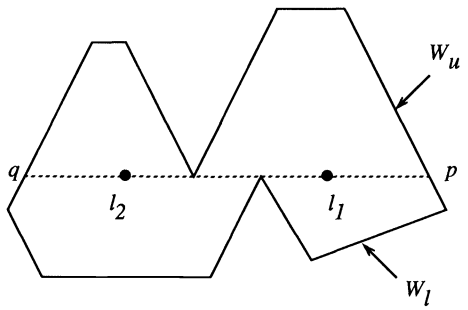


FIG. 15. $\overline{l_1 l_2} \cap W_u \neq \emptyset$ *and* $\overline{l_1 l_2} \cap W_l \neq \emptyset$.

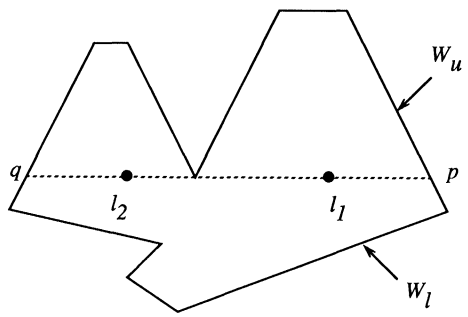


FIG. 16. $\overline{l_1 l_2} \cap W_u \neq \emptyset$ *and* $W_l \subseteq V_{l_1}$.

$W_l \not\subseteq V_{l_i}$ for $i = 1, 2$. The argument for the other case ($\overline{l_1 l_2} \cap W_u = \emptyset$, $\overline{l_1 l_2} \cap W_l \neq \emptyset$, and $W_u \not\subseteq V_{l_i}$ for $i = 1, 2$) is similar and is thus omitted.

Since $l_1, l_2 \notin b(P)$ and $\overline{l_1 l_2} \cap W_u \neq \emptyset$, there exist maximal open segments $(a_1, b_1)_{b(P)} \subseteq W_u - V_{l_2}$ and $(a_2, b_2)_{b(P)} \subseteq W_u - V_{l_1}$ not visible from l_2 and l_1, respectively, such that a_1, b_1, a_2, and b_2 appear in counterclockwise order in W_u (Fig. 17). By condition (P1), if $b_1 \neq a_2$ then $\overline{b_1 a_2} \subseteq b(P)$. For $i = 1, 2$, let R_i be the closed simple region whose boundary is $[a_i, b_i]_{b(P)} \cup \overline{b_i a_i}$. Let R_0 be the closed region whose boundary is $W_l \cup \overline{pq}$.

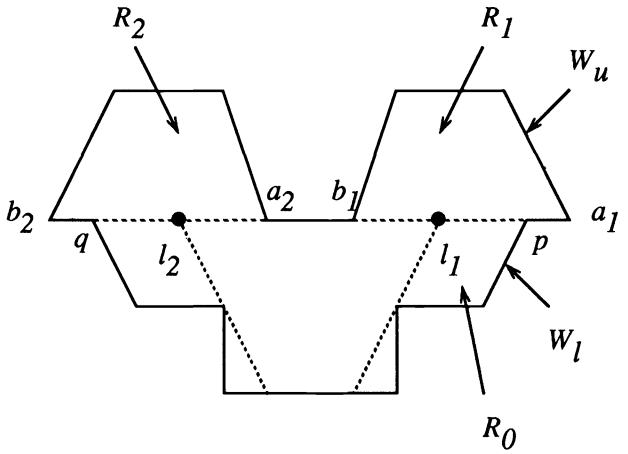


FIG. 17. *Illustration for the proof of Lemma 5.*

Assume that there exists a search schedule for P, and let F be a search schedule in which the total number of times R_1 and R_2 are changed from contaminated to clear is smallest among all search schedules. First, as we did in the proof of Lemma 4, we can show that R_1 and R_2 cannot be cleared simultaneously. Thus without loss of generality we can assume that R_1 is changed from contaminated to clear at t_1, R_1 remains clear after t_1, and R_2 is contaminated at t_1 or at some time after t_1. Let $t_2 > t_1$ be the first time after t_1 at which R_2 is changed from contaminated to clear. Then by $\overline{l_1 l_2} \cap W_l = \emptyset$, the assumption on F and an argument similar to that in the proof of Lemma 4, we can show that R_0 and R_2 are contaminated at t_1 (more specifically, any point in $R_0 \cup R_2$ which is not illuminated is contaminated at t_1). Next, by using the assumption that $\overline{l_1 l_2} \cap W_l = \emptyset$, we can show that R_0 must be clear at $t_2 - \delta$ for some $\delta > 0$.

In summary, R_1 is clear in $[t_1, t_2]$, R_2 is contaminated in $[t_1, t_2)$, R_0 is contaminated at t_1, and R_0 is changed from contaminated to clear in $[t_1, t_2)$. Since by assumption $W_l \not\subseteq V_{l_i}$, for $i = 1, 2$, R_0 cannot be cleared unless each of l_1 and l_2 is aimed at the points in W_l not visible from the other searchlight. Here, since R_1 is clear and R_2 is contaminated in $[t_1, t_2)$, l_2 must be aimed at a_1 and b_1 whenever l_1 is aimed at a point W_l not visible from l_2 (Fig. 18), and l_1 must be aimed at a_2 and b_2 whenever l_2 is aimed at a point in W_l not visible from l_1 (Fig. 19). Thus (1) at any time in $[t_1, t_2)$ at most one of l_1 and l_2 can be aimed at a point in W_l not visible from the other searchlight, and (2) if l_1 and l_2 (or l_2 and l_1) are aimed at a point in W_l not visible from the other searchlight at s_1 and s_2 for some $t_1 \leqq s_1 < s_2 < t_2$, respectively, then there exists some $s_1 < t < s_2$ such that any two points in W_l visible from only one of l_1 and l_2 are nonseparable at t. This observation, together with Lemma 1 and the fact that any point in R_0 which is not illuminated is contaminated at t_1, implies that R_0 contains a

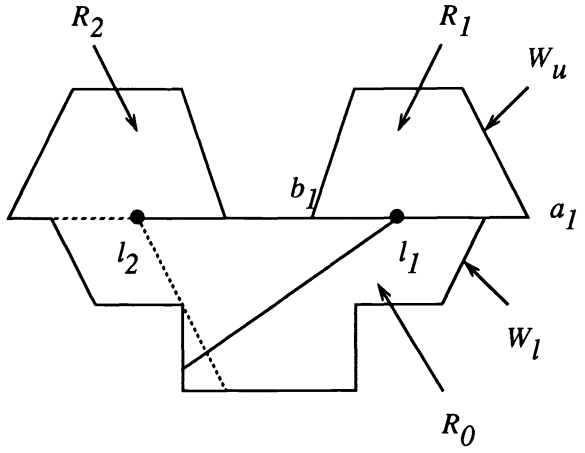


FIG. 18. *Illustration for the proof of Lemma* 5; l_1 *is aimed at a point in* W_l *not visible from* l_2.

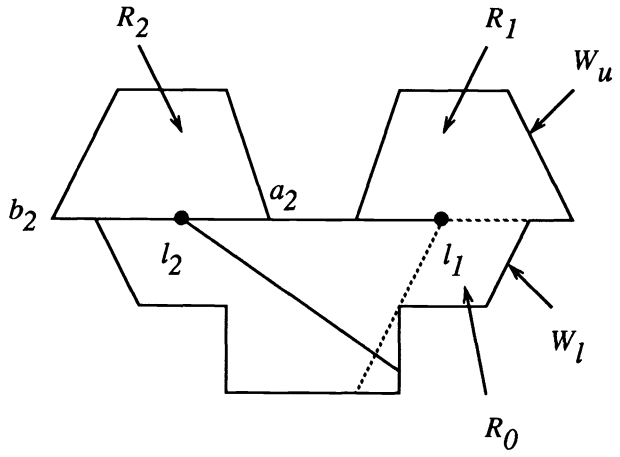


FIG. 19. *Illustration for the proof of Lemma* 5; l_2 *is aimed at a point in* W_l *not visible from* l_1.

contaminated point at any time in $[t_1, t_2)$, and hence R_0 cannot be clear in $[t_1, t_2)$. This is a contradiction. □

Example 4. Consider the instance $S = (P, \{l_1, l_2, l_3\})$ shown in Fig. 20. When the one-way sweep strategy is applied to S, we obtain a semiconvex subpolygon Q of P supported by $\{l_1\}$ containing two searchlights l_2 and l_3 in the interior. Note that the strategy cannot be applied to Q, since the interior of Q is visible only from l_2 and l_3. Also, the instance $S_Q = (Q, \{l_2, l_3\})$ does not satisfy any of the conditions of Theorem 5, and hence there exists no search schedule for S_Q. Thus by Theorem 1, there exists no search schedule for S.

6. Concluding remarks. We have posed the searchlight scheduling problem and presented various conditions for the existence of a search schedule. In particular, we have shown that the problem of obtaining a search schedule for an instance having at least one searchlight on the polygon boundary can be reduced to that for instances having no searchlight on the polygon boundary, and then presented a simple necessary and sufficient condition for the existence of a search schedule for instances having exactly two searchlights in the interior. Some preliminary results for the case in which

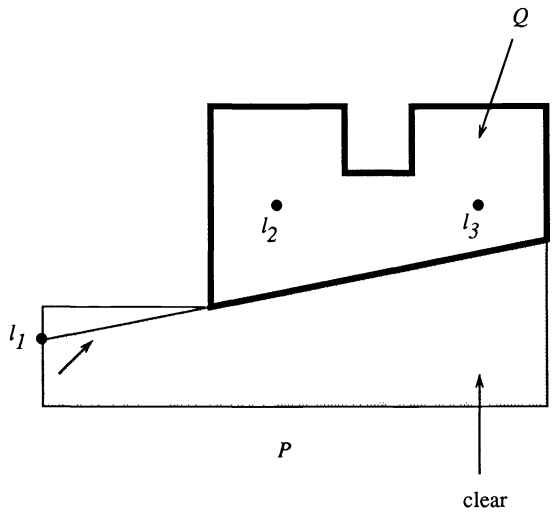


FIG. 20. *An instance having no search schedule.*

there are three searchlights in the interior have been reported in [8], but obtaining a necessary and sufficient condition for this case remains as a challenging open problem.

As a final note, we remark that given an n-sided simple polygon P we can compute, in $O(n \log\log n)$ time, a set L of searchlights such that (1) $|L| = \lfloor n/3 \rfloor$ and (2) the instance $S = (P, L)$ has a search schedule. This is an immediate corollary of Theorem 2 and a linear time coloring algorithm (see [1], [6, Chap. 1]) for computing, given a triangulation of P, a subset L of the vertices of P such that $L = \lfloor n/3 \rfloor$ and every point in the interior of P is visible from at least one vertex in L. It is known that a triangulation of an n-sided polygon can be computed in $O(n \log\log n)$ time [9]. If P is rectilinear, then a set L with the desired property such that $|L| = \lfloor n/4 \rfloor$ can be computed in $O(n \log\log n)$ time [7].

Acknowledgments. We wish to thank the anonymous referees for their careful reading of and helpful comments on this paper.

REFERENCES

[1] D. AVIS AND G. T. TOUSSAINT, *An efficient algorithm for decomposing a polygon into star-shaped polygons*, Pattern Recognition, 13 (1981), pp. 395-398.

[2] V. CHVÁTAL, *A combinatorial theorem in plane geometry*, J. Combin. Theory Ser. B, 18 (1975), pp. 39-41.

[3] J. KAHN, M. KLAWE, AND D. KLEITMAN, *Traditional galleries require fewer watchmen*, SIAM J. Algebraic Discrete Methods, 4 (1983), pp. 194-206.

[4] D. T. LEE AND A. K. LIN, *Computational complexity of art gallery problems*, IEEE Trans. Inform. Theory, 32 (1986), pp. 276-282.

[5] J. O'ROURKE, *An alternate proof of the rectilinear art gallery theorem*, J. Geometry, 21 (1983), pp. 118-130.

[6] ———, *Art Gallery Theorems and Algorithms*, Oxford University Press, New York, 1987.

[7] J.-R. SACK AND G. TOUSSAINT, *Guard placement in rectilinear polygons*, Tech. Report, School of Computer Science, McGill University, Montreal, Quebec, Canada, February 1988.

[8] K. SUGIHARA, I. SUZUKI, AND M. YAMASHITA, *The searchlight scheduling problem*, Tech. Report, Department of Electrical Engineering and Computer Science, University of Wisconsin-Milwaukee, Milwaukee, Wisconsin, October 1988.

[9] R. E. TARJAN AND C. J. VAN WYK, *An $O(n \log\log n)$-time algorithm for triangulating a simple polygon*, SIAM J. Comput., 17 (1988), pp. 143-178.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1041-1050, December 1990

005

MINIMUM CUTS FOR CIRCULAR-ARC GRAPHS*

D. T. LEE†, M. SARRAFZADEH†, AND Y. F. WU†‡

Abstract. The problem of finding a minimum cut of n arcs on a unit circle is considered. It is shown that this problem can be solved in $\Theta(n \log n)$ time, which is optimal to within a constant factor. If the endpoints of the arcs are sorted, the problem can be solved in linear time. The solution to the minimum cut problem can be used to solve a minimum new facility problem in competitive location and a minimum partition set problem for the intersection model of a circle graph. As a by-product it is also shown that the maximum independent set of n arcs can be obtained in linear time, assuming the endpoints are sorted, which is much simpler than the most recent result of Masuda and Nakajima [*SIAM J. Comput.*, 17 (1988), pp. 41-52].

Key words. computational complexity, algebraic computation trees, circular-arc graphs, circle graphs, minimum covering, maximum independent set

AMS(MOS) subject classifications. 68Q25, 68Q20, 68R10, 90B50

1. Introduction.

We consider the following problem:

Problem 1 (Minimum cuts for circular arcs (MCCA)). Given a set S of n nonempty open arcs $S = \{A_1, A_2, \cdots, A_n\}$ on a unit circle $\mathcal{K}$, find a set C of *cut points* $C = \{c_1, c_2, \cdots, c_m\}$, for some $m > 0$, such that each arc A_i contains at least one cut point and the cardinality of C is minimized.

The number m is referred to as a *minimum cut number* of S, and C is referred to as the *minimum cut set* of S. For convenience each arc A_i, $i = 1, 2, \cdots, n$ is represented as $(\theta_{ib}, \theta_{ie})$, where θ_{ib} and θ_{ie} denote, respectively, the beginning and ending points of the arc when it is traversed in counterclockwise manner, starting with an arbitrarily chosen point on $\mathcal{K}$ that is not an endpoint of any arc in S.

Before we give details of how to solve the minimum cut problem for circular-arcs optimally, we consider two applications that arise in different contexts.

Consider the so-called *one-on-one competitive location problem* studied by Drezner [2]: Given a set of demand points p_i with associated weight w_i, $i = 1, 2, \cdots, n$, in the plane and an existing facility X, locate a new facility Y, so that Y will attract the most total weight; a demand point p_i (and its associated weight w_i) is considered attracted to Y if $d(p_i, Y) < d(p_i, X)$, where $d(a, b)$ is the Euclidean distance between points a and b. This problem was solved optimally in $\Theta(n \log n)$ time [2], [4]. A related problem, the *Minimum New Facilities problem*, is of interest here and is defined below. For a more general description of competitive location problems, see [3].

Problem 2 (Minimum new facilities (MNF)). Given an existing facility X serving n demand points $p_i = (x_i, y_i)$, $i = 1, 2, \cdots, n$, find a minimum number m of new facilities at least R away from X so as to attract all demand points that can be attracted; a demand point p_i is considered attracted by a new facility Y if $d(p_i, Y) < d(p_i, X)$.

Without loss of generality, we can assume that the existing facility is at the origin. Let $r = R/2$. Denote the circle of radius R centered at the origin as the *R-circle*, and

* Received by the editors December 19, 1988; accepted for publication (in revised form) February 14, 1990. This work was supported in part by the National Science Foundation under grants DCR 84-20814 and MIP 87-09074.

† Department of Electrical Engineering and Computer Science, Northwestern University, Evanston, Illinois 60208.

‡ Present address, MCC VLSI-CAD, 3500 West Balcones Center Drive, Austin, Texas 78759.

denote the circle of radius r centered at the origin as the *r-circle.* Although the position of each demand point is specified in terms of its Cartesian coordinates, in the course of the following discussion, we shall use polar coordinates for convenience, i.e., $p_i = (r_i, \theta_i)$, $i = 1, 2, \cdots, n$, where r_i is the polar radius and θ_i is the polar angle. If there is a demand point p_i located within the r-circle, i.e., $r_i \leqq r$, then it is impossible to attract p_i away from the existing facility using any facility at least R away from the origin. Therefore, we shall assume from now on that every $r_i > r$.

For any facility located more than R away from the origin, we can always shift it closer to the origin without losing the attraction of any demand points (in fact, doing so may attract more demand points). Thus we can restrict ourselves to facilities on the R-circle only. Given a demand point p_i, the range of angles in which we can locate a new facility on the R-circle to attract p_i is $A_i = (\theta_{ib}, \theta_{ie}) = (\theta_i - \cos^{-1}(r/r_i), \theta_i + \cos^{-1}(r/r_i))$, where θ_{ib} and θ_{ie} are determined by the tangent points of the two tangents of the r-circle passing through p_i (see Fig. 1). We treat A_i as an open arc on the R-circle and call it a *demand arc* with the following implication: if there is any facility located on this open arc, then demand point p_i is attracted.

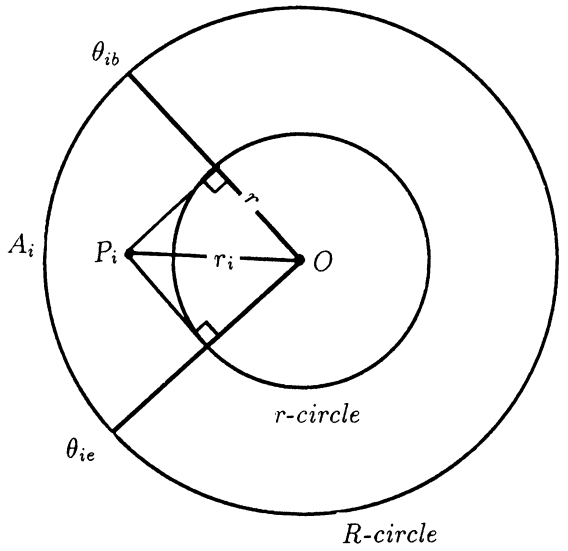


FIG. 1. *Mapping of a demand point to a demand arc.*

Through these assumptions and mapping every demand point to its corresponding demand arc, we have transformed the MNF problem into an instance of the MCCA problem with the following correspondence: the demand arcs are the input arcs to the MCCA, and the locations of the new facilities correspond to the positions of the cut points. Note that the values of the angles of demand arcs are taken modulo 2π, i.e., each arc is of length less than 2π.

Next we consider a partition problem that arises in topological via minimization [6].

Problem 3 (Minimum partition set (MPS)). Given a set $S = \{u_1, u_2, \cdots, u_n\}$, of n chords in a unit circle $\mathcal{K}$, find a *partition set* $P = \{p_1, p_2, \cdots, p_K\}$ of points on $\mathcal{K}$ such that no chord in S has both of its endpoints on the same circular-arc defined by two consecutive points, p_i and p_{i+1}, for $i = 1, 2, \cdots, K$, where $p_{K+1} = p_1$ and K is minimum.

The set S of chords induces the intersection model of the so called *circle graph* of n chords. The cardinality K of the minimum partition set P is called the *partition number* of (chord) representation of a circle graph. The resulting partition of the endpoints of the chords (into K subsets) has the following property: the (intersection of the) subset of chords with an endpoint in the same partition defines a *permutation graph.* Recall that a set T of chords forms a permutation graph if there exists a chord (not in T) that cuts across each chord in T. Of interest here is to determine the partition number of representation of any circle graph, for it reflects the degree of difficulty in finding a maximum 2-independent set of the circle graph. (A 2-independent set of chords is a set that can be partitioned into two subsets such that no two chords in each subset intersect. For details, the reader is referred to [6].)

The problem MPS can be transformed into an instance of MCCA as described below. Assume that each chord u_t is represented as a pair of integers (t_i, t_j), where t_i and t_j represent the *rank* of the endpoints sorted in counterclockwise direction with respect to an arbitrary point on $\mathcal{K}$. Now for each chord u_t we construct two arcs, A_t^1 and A_t^2: $A_t^1 = (t_i + \delta, t_j - \delta)$ and $A_t^2 = (t_j + \delta, t_i - \delta)$, where δ represents a small deviation from the corresponding endpoints. It is obvious to see that to separate the two endpoints of each chord u_t, it is sufficient to place at least a *cut-point* on each of its two associated arcs A_t^1 and A_t^2. Thus we have an instance of MCCA in which these $2n$ arcs are the input arcs, and the *minimum cut set* corresponds to the *partition set* of the circle graph.

2. Preliminaries. We now begin our discussion of how to solve the MCCA problem. We begin with some definitions. Recall that we have a set S of arcs, $\{A_1, A_2, \cdots, A_n\}$, each of which is represented as $(\theta_{ib}, \theta_{ie})$, where θ_{ib} and θ_{ie} denote, respectively, the beginning and ending points of the arc when it is traversed in counterclockwise manner. The endpoints, θ_{ib} and θ_{ie}, of A_i are called CW (*for clockwise*) *end* and CCW (*for counterclockwise*) *end*, respectively. Let a *cut* at angle α denote a half-line emanating from the origin with orientation α.

DEFINITION 1. A *cut interval* is a nonempty angular interval $c = (\theta_b, \theta_e)$ or $[\theta_b, \theta_e)$ such that $\theta_e \in \{\theta_{ie} \mid i = 1, \cdots, n\}$ and $\theta_b \in \{\theta_{ib} \mid i = 1, 2, \cdots, n\} \cup \{\theta_{ie} \mid i = 1, 2, \cdots, n\}$, and for all $\theta_1, \theta_2 \in c$, the set of arcs intersecting the cut at θ_1 is the same as the set of arcs intersecting the cut at θ_2; c does not contain θ_b, if and only if $\theta_b \in \{\theta_{ib} \mid i = 1, 2, \cdots, n\}$, i.e., θ_b is a CW end.

In general, each θ_{ie} (or arc A_i) *determines* a cut interval; thus we have n cut intervals unless some of the θ_{ie}'s of the n given arcs are not distinct. Let us denote the set of $t \leq n$ cut intervals as $\mathcal{C} = \{c_1, \cdots, c_t\}$. If the endpoints are not distinct, we shall use the following convention that *beginning* endpoints always come *before ending* points, and for cases where two beginning (or similarly, ending) points of different arcs coincide, the arc with smaller index comes first. In other words, when sorting of these n arcs is performed, it is done in lexicographic ascending order of *angle, type of endpoint, index*, where type of endpoint is either *b* or *e*, denoting *beginning* or *ending*, respectively. We shall use the phrase "arc A_i is *covered* by cut interval c_j" to mean that any cut point located within c_j intersects A_i. Figure 2 shows an example with arcs $A_1, \cdots, A_8$, and cut intervals $c_1, \cdots, c_7$; c_1 is open on both ends, while c_7 is closed on the CW end and open on the CCW end.

LEMMA 1. *A minimum number of cuts that cover all arcs can be obtained by placing at most one cut inside each cut interval and none outside the union of all cut intervals.*

Proof. In an optimal placement of cuts, it there is a cut at angle α that does not lie in any cut interval, we can always shift the cut CCW until it reaches a cut interval, say at angle α'. The new cut still intersects every arc that intersects the old cut because

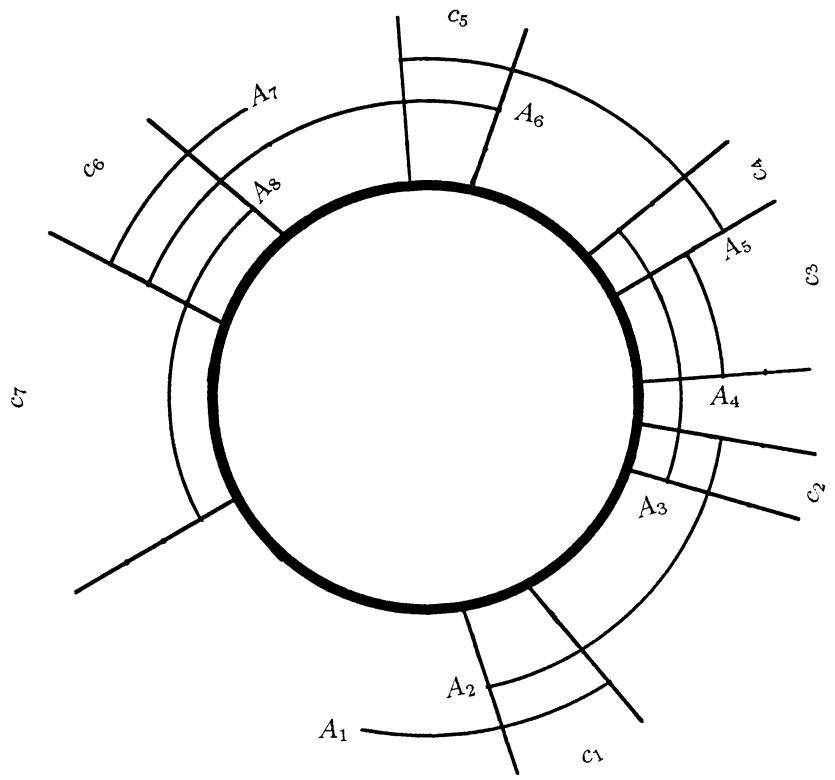


FIG. 2. *Illustration of cut intervals.*

the shifting process does not move the cut away from any of these arcs. Now if there are already some cuts in the new cut interval, we can arbitrarily retain one cut and eliminate the rest and still keep the original intersections. Thus our restriction does not prevent us from finding an optimal placement of cuts. □

DEFINITION 2. The *successor interval function* $\mathcal{S}$ maps a cut interval in $\mathcal{C}$ or ϕ into another cut interval or ϕ in the following way: $\mathcal{S}(\phi) = \phi$; for all $c_i \in \mathcal{C}$, $\mathcal{S}(c_i) = \phi$, if there exists no arc not covered by c_i. Otherwise, $\mathcal{S}(c_i) = c_j$, where c_j is the first cut interval encountered when traversing the circle CCW from c_i such that its CCW end is determined by an arc not covered by c_i.

We use the abbreviation $\mathcal{S}^j(c_i)$ to denote $\overbrace{\mathcal{S}(\mathcal{S}(\cdots \mathcal{S}(c_i))\cdots)}^{j}$. The following lemma is obvious.

LEMMA 2. *If there exists a cut interval c_i such that $\mathcal{S}(c_i) = \phi$, then the minimum cut set $\mathcal{C} = \{c_i\}$.*

Hereafter, we shall consider nontrivial cases where $\mathcal{S}(c_i) \neq \phi$, for all $c_i \in \mathcal{C}$. From Lemma 1 we may consider each cut interval c_i as a *cut point* cp_i on the circle with angle $\theta_i = \frac{1}{2}(\theta_{ib} + \theta_{ie})$ and shall use the terms, cut interval and cut point, interchangeably. In the subsequent figures, a cut interval is indicated by its cut point.

DEFINITION 3. The *angle* α *spanned* by a sequence of cut intervals, $(c_i, \mathcal{S}(c_i), \cdots, \mathcal{S}^j(c_i))$ is defined as the *angular displacement* from the cut point cp_i to the cut point corresponding to $\mathcal{S}^j(c_i)$.

We use "$\leftrightarrow$" to denote the relative positioning of cut intervals on the circle in the following way: $c_i \leftrightarrow c_j \leftrightarrow c_k \leftrightarrow \cdots \leftrightarrow c_l$ means that the cut intervals $c_i, c_j, c_k, \cdots, c_l$ occur

in counterclockwise order such that c_j follows c_i, c_k follows $c_j, \cdots$, and c_i follows c_l. The successor interval function on the cut intervals has the following *monotonicity property.*

LEMMA 3. *Consider any two cut intervals c_i and c_j. Let $c_k = \mathcal{S}(c_i)$ and $c_l = \mathcal{S}(c_j)$. If $c_i \leftrightarrow c_j \leftrightarrow c_k$, then either $c_l = c_k$ or $c_k \leftrightarrow c_l \leftrightarrow c_i$.*

Proof. It suffices to show that it is not possible to have $c_j \leftrightarrow c_l \leftrightarrow c_k$. Suppose it were. The arc A, which determines $c_l = \mathcal{S}(c_j)$, is not covered by c_j by the definition of the function $\mathcal{S}$. Clearly, the arc A is not covered by c_i. Then $\mathcal{S}(c_i)$ would be c_l, a contradiction to the assumption that $\mathcal{S}(c_i) = c_k$. □

DEFINITION 4. A *Minimal Covering (cut interval)*[1] *Sequence* (MCS) starting with c_i is a sequence of cut intervals, denoted $\text{MCS}(c_i) = (c'_1, c'_2, \cdots, c'_j)$, such that the following hold:

(a) $c'_1 = c_i$,

(b) $c'_{l+1} = \mathcal{S}(c'_l)$, for $1 \leqq l \leqq j$,

(c) j is the *smallest* integer so that the angle (cf. Definition 3) spanned by $(c'_1, c'_2, \cdots, c'_j, c'_{j+1})$ is greater than or equal to 2π, where $c'_{j+1} = \mathcal{S}(c'_j)$.

It is easy to see that an MCS starting with any cut interval c_i covers the entire set of arcs in S.

LEMMA 4. *Suppose that a cut must be placed within the cut interval c_i. Then an optimal solution to this restricted* MCCA *problem is to place one cut within each cut interval in the minimal covering sequence* $\text{MCS}(c_i)$.

Proof. The suggested placement is greedy in nature. Suppose we have placed one cut within each of c_i, $\mathcal{S}(c_i)$, $\mathcal{S}^2(c_i), \cdots, \mathcal{S}^k(c_i)$ and there are still arcs not covered by any cut. Then obviously the best way to place the next cut is to place it within $\mathcal{S}^{k+1}(c_i)$ because we have to cover the arc A that determines $\mathcal{S}^{k+1}(c_i)$ and is not covered by $\mathcal{S}^k(c_i)$, and placing it at any point between $\mathcal{S}^k(c_i)$ and $\mathcal{S}^{k+1}(c_i)$ will not do any better. □

DEFINITION 5. Given an arbitrary cut interval $c_j \in C$ where $1 \leqq j \leqq t$, define $\mathcal{Q}_j$, $\mathcal{Q}_{j0}$, $\mathcal{Q}_{j*}$ as follows.

$\mathcal{Q}_j \equiv$ the infinite sequence of cut intervals $(c_j, \mathcal{S}(c_j), \mathcal{S}^2(c_j), \mathcal{S}^3(c_j), \cdots)$,

$\mathcal{Q}_{j0} \equiv (c'_1 = c_j, c'_2, \cdots, c'_k, \cdots, c'_l)$, is a finite subsequence of $\mathcal{Q}_j$, and

$\mathcal{Q}_{j*} \equiv (c'_k, \cdots, c'_l)$ is a subsequence of $\mathcal{Q}_{j0}$, such that

(1) $c'_i = \mathcal{S}(c'_{i-1})$ for $1 < i \leqq l$, and $c'_k = \mathcal{S}(c'_l)$, and

(2) all elements in $\mathcal{Q}_{j0}$ are distinct.

In other words, $\mathcal{Q}_{j0}$ is the longest nonrepeating subsequence of $\mathcal{Q}_j$, and $\mathcal{Q}_{j*}$ is the repeating subsequence of $\mathcal{Q}_j$.

LEMMA 5. *Let $|\text{MCS}(c_i)|$ denote the number of cut intervals in the minimal cut sequence starting with c_i. Then $|\text{MCS}(c'_u)| = |\text{MCS}(c'_v)|$ for all c'_u, $c'_v \in \mathcal{Q}_{j*}$, and $u \neq v$.*

Proof. Let $\mathcal{Q}_{j*} = (c'_1, c'_2, \cdots, c'_q)$. Recall that $c'_i = \mathcal{S}(c'_{i-1})$ for $i = 2, 3, \cdots, q$ and $\mathcal{S}(c'_q) = c'_1$. We distinguish two cases.

Case 1. Suppose $\mathcal{Q}_{j*} = \text{MCS}(c'_1)$, i.e., the angle spanned by $(c'_1, c'_2, \cdots, c'_q, c'_{q+1})$, where $c'_{q+1} = c'_1$, is exactly 2π. See Fig. 3(a). Since we have a "circular" sequence, the claim follows.

Case 2. The angle spanned by $(c'_1, c'_2, \cdots, c'_q, c'_{q+1})$ is greater than 2π. See Fig. 3(b). Let $\mathscr{C}$ denote the MCS (c'_1), i.e., $\mathscr{C} = (c'_1, c'_2, \cdots, c'_m)$ for some $m < q$. We claim that $|\text{MCS}(c'_s)| = m$ for any cut interval $c'_s \in \mathcal{Q}_{j*}$. From Lemma 3, it follows that $c'_{m+1} = \mathcal{S}(c'_m)$ must fall between intervals c'_1 and c'_2, c'_{m+2} between intervals c'_2 and c'_3, and so on. Starting with interval c'_2, the MCS (c'_2) has exactly m cut intervals. Thus, $|\text{MCS}(c'_1)| = |\text{MCS}(c'_2)|$. The claim follows by repeating the arguments.

[1] We shall omit the phrase "cut interval" in subsequent discussions.

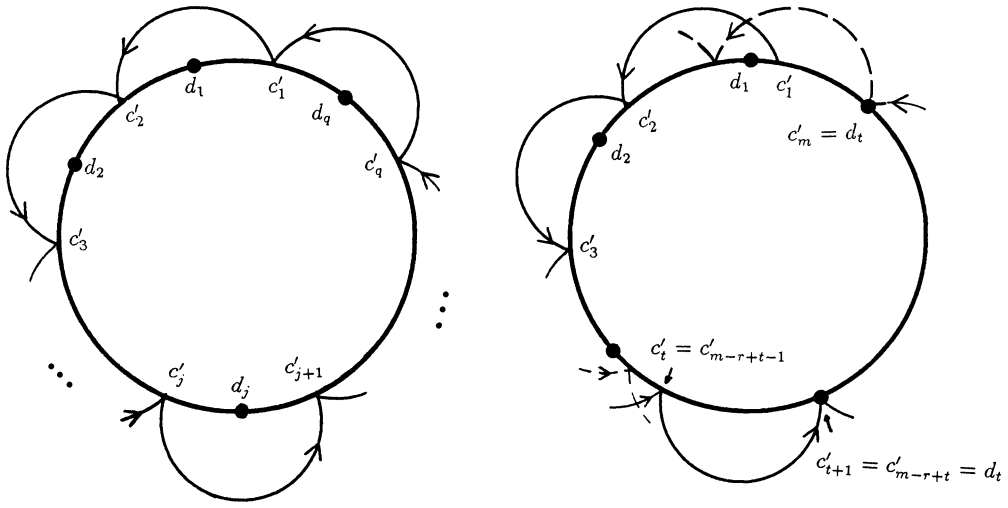


FIG. 3. *Proof of Lemma* 5.

THEOREM 1. *The* MCCA *problem, as defined earlier, can be solved by placing one cut at each cut interval in the minimal cover sequence starting at any cut interval* c'_s, *i.e.,* MCS (c'_s), *for* $c'_s \in \mathcal{Q}_{j*}$.

Proof. From Lemma 5 we may consider only MCS (c'_1) (where $\mathcal{Q}_{j*} = (c'_1, c'_2, \cdots, c'_q)$). As in Lemma 5, we distinguish two cases.

Case 1. The angle spanned by $(c'_1, c'_2, \cdots, c'_q, c'_{q+1})$ is exactly 2π. See Fig. 3(a). It is obvious that q cut points are necessary, as there are q distinct nonoverlapping arcs that need to be covered.

Case 2. The angle spanned by $(c'_1, c'_2, \cdots, c'_q, c'_{q+1})$ is greater than 2π. See Fig. 3(b). Let $\mathscr{C}$ denote the MCS (c'_1), i.e., $\mathscr{C} = (c'_1, c'_2, \cdots, c'_m)$ for some $m < q$. We claim that the solution to the MCCA problem has exactly m cut points. Note that from Lemma 3, $c'_{m+1} = \mathscr{S}(c'_m)$ must fall between intervals c'_1 and c'_2, c'_{m+2} between intervals c'_2 and c'_3, and so on. Assume that there exists a solution $D = \{d_1, d_2, \cdots, d_r\}$ with $r < m$ and $d_1 \leftrightarrow d_2 \leftrightarrow \cdots \leftrightarrow d_r$. We distinguish three subcases:

Subcase 1. $D \subset Q_{j*}$. In this case D must be an MCS starting with, say, d_1, according to Lemma 4. By Lemma 5 all MCSs are of the same size; it is, therefore, impossible to have a covering with $r < m$.

Subcase 2. $\exists t$, $1 < t \leqq r$, such that $\{d_1, \cdots, d_{t-1}\} \cap \mathcal{Q}_{j*} = \phi$, and $\{d_t, \cdots, d_r\} \subset \mathcal{Q}_{j*}$. Since $\{d_t, \cdots, d_r\} \subset \mathcal{Q}_{j*}$, it must be a sequence induced by the successor interval function $\mathscr{S}$. Therefore we can reindex the cut intervals in $\mathcal{Q}_{j*}$ and select an MCS so that the last $r-t+1$ cut intervals match those of D. Let us assume that the MCS is MCS (c'_1) with $d_{t+i} = c'_{m-r+t+i}$, for $0 \leqq i \leqq r-t$. Now consider the set of intervals $\mathscr{R}$ defined by the cut points corresponding to the first $m-r+t-2$ cut intervals in MCS (c'_1), i.e., $\mathscr{R} = \{(cp'_1, cp'_2), (cp'_2, cp'_3), \cdots, (cp'_{m-r+t-2}, cp'_{m-r+t-1})\}$. From the assumption that $r < m$, $|\mathscr{R}| \geqq t-1$. That is, there are no fewer than $t-1$ intervals that must contain exactly $t-1$ d_i's in D, $1 \leqq i < t$. Clearly the only possibility is that $r = m-1$, and one d_i is placed in each interval in $\mathscr{R}$, as shown in Fig. 3(b). Note that the interval $(cp'_{m-r+t-1}, cp'_{m-r+t}) = (cp'_t, cp'_{t+1})$, for $r = m-1$, contains no element of D. Now let us focus on the cut interval following c'_m in $\mathcal{Q}_{j*}$. c'_{m+1} must fall between d_1 and cp'_2; otherwise the arc determining c'_{m+1} will not be covered by D. Similarly, c'_{m+2},

$c'_{m+3}, \cdots, c'_{m+t-1}$ must fall between appropriate intervals. A contradiction results when considering the next cut interval, c'_{m+t}, which must lie between c'_t and c'_{t+1}, and (cp'_t, cp'_{t+1}) contains no element of D.

Subcase 3. $D \cap \mathcal{Q}_{j*} = \phi$. Let $\mathcal{C}$ denote MCS (c'_1), i.e., $\mathcal{C} = \{c'_1, c'_2, \cdots, c'_m\}$, with c'_{m+1} lying between c'_1 and c'_2. Let $\mathcal{R}$ denote the set of corresponding cut points, i.e., $\mathcal{R} = \{cp'_1, cp'_2, \cdots, cp'_m\}$. All the elements in D must fall in the intervals (cp'_i, cp'_{i+1}), $i = 1, 2, \cdots, m$, where $cp'_{m+1} = cp'_1$, with the condition that each element falls in one interval, except possibly the interval (cp'_m, cp'_1). (Recall that $r = m - 1$.) (See Fig. 4.) By similar arguments as in Subcase 2, the cut interval c'_{m+m} will result in a contradiction. This completes the proof. □

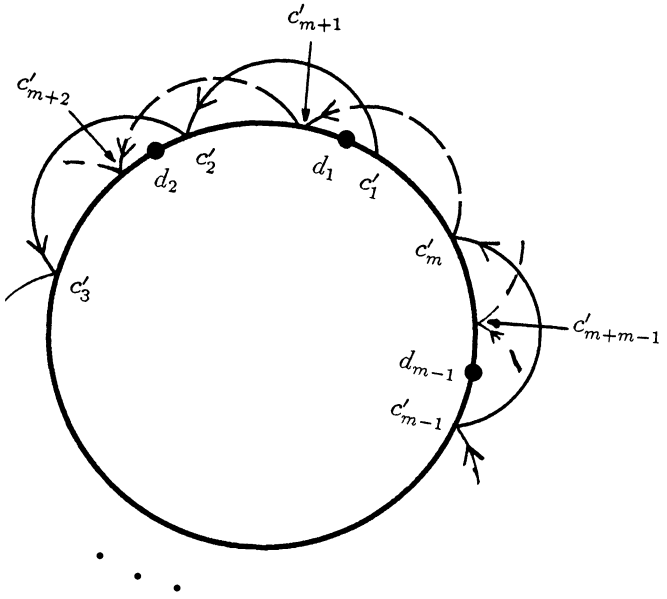


FIG. 4. *Illustration for the proof of Theorem* 1.

3. The algorithm. Based on our discussions above, we can now describe the algorithm for the MCCA problem.

ALGORITHM MCCA

Input. A set S of n circular-arcs, $S = \{A_1, A_2, \cdots, A_n\}$, on a unit circle centered at the origin O. Each arc is represented as $A_i = (\theta_{ib}, \theta_{ie})$, where θ_{ib} and θ_{ie} denote, respectively, the angles of the beginning and ending points of arc A_i with respect to the origin and the positive x-axis when it is traversed in counterclockwise manner.

Output. A set $\mathcal{C}$ of cut points of minimum cardinality so that each arc in S contains at least one point in $\mathcal{C}$.

Begin

(1) Sort the endpoints of the arcs in the counterclockwise direction and arrange them in a circular list $\mathcal{L}$.

(2) Compute the set of cut intervals C by traversing $\mathcal{L}$.

(3) Compute the successor interval function $\mathcal{S}$ for $\mathcal{C}$.

(4) If $\mathcal{S}(c_i) = \phi$ for some $c_i \in \mathcal{C}$, then return $\mathcal{C} = \{c_i\}$.
(5) Pick an arbitrary cut interval c from C, form the sequence c, $\mathcal{S}(c)$, $\mathcal{S}^2(c), \cdots$, and compute the repeating subsequence $\mathcal{Q}* = (c_{j_1}, c_{j_2}, \cdots, c_{j_r})$.
(6) Select an arbitrary cut interval, say $c_{j_1} \in \mathcal{Q}*$, and find the MCS (c_{j_1}), i.e., find the smallest index m such that the angle α, spanned by c_{j_1}, $c_{j_2}, \cdots, c_{j_{m+1}}$, is greater than or equal to 2π.
(7) Return $\mathcal{C} = \{c_{j_1}, c_{j_2}, \cdots, c_{j_m}\}$.

end

THEOREM 2. *Algorithm* MCCA *takes* $\Theta(n \log n)$ *time and is optimal. Furthermore, if the endpoints are sorted,* $O(n)$ *time suffices.*

Proof. The correctness of the algorithm follows from the above discussion. The time bound is due to sorting of the endpoints (*step* 1). It is not difficult to see that each of the remaining steps takes only $O(n)$ time. The algorithm is optimal because we can reduce the *Minimum Gap problem*, which requires $\Omega(n \log n)$ time under the algebraic computation tree model of Ben-Or [1], to the MCCA problem. The reduction is as follows.

An instance of the Minimum Gap problem follows:

Given $x_1, x_2, \cdots, x_n \in R$ and $\varepsilon > 0$, determine if $|x_i - x_j| \geqq \varepsilon$ for all $i \neq j$.

We first find the maximum, $x_{\max}$, and the minimum, $x_{\min}$, of the n numbers and map every number x_i into an open arc $((x_i - x_{\min})/(x_{\max} - x_{\min} + \varepsilon), (x_i - x_{\min} + \varepsilon)/(x_{\max} - x_{\min} + \varepsilon))$ on the unit circle. Now we invoke algorithm MCCA to compute the minimum number of cuts needed to cover all arcs. If the answer is n, answer YES to the original problem; otherwise answer NO. Clearly the transformation is done in $O(n)$ time, thus establishing the $\Omega(n \log n)$ lower bound for the MCCA problem. □

Remark. The above reduction is based on the assumption that in the input, each arc is specified in terms of the radiant values of its two end angles. Thus to solve the Minimum New Facilities problem using algorithm MCCA, we have to use inverse trigonometric functions, which are not available in the algebraic computation tree model, to compute angles from the x and y coordinates of the given demand points. In fact, we can avoid trigonometric functions by using the intersection point between the cut at angle α and the unit circle, called the *representing point* of α, to specify α. For example, $(\sqrt{3}/2, 1/2)$ is the representing point of the angle $\pi/6$. Given a demand point p_i, the representing points of the beginning and ending angles of A_i, p_i's demand arc, can be readily computed using operations in the algebraic computation tree model. Let's refer to the new MCCA problem, using representing points as the input/output format, as the MCCA′ problem. The new MCCA algorithm, denoted as algorithm MCCA′, is almost identical to algorithm MCCA because the comparison of two angular positions can still be done in constant time.

THEOREM 3. *Both* MCCA′ *and* MNF *problems require* $\Omega(n \log n)$ *time.*

Proof. We show the proof by using a reduction from the *Minimum Gap on a Circle problem.* The Minimum Gap on a Circle problem follows:

Given n points $(x_1, y_1), (x_2, y_2), \cdots, (x_n, y_n)$ on the first quadrant of the unit circle, and $\varepsilon > 0$, determine if the straight line distance between any pair of points is at least ε.

We can show that the Minimum Gap on a Circle problem requires $\Omega(n \log n)$ time under the algebraic computation tree model, using the same approach as in the lower bound proof for the *Maximum Gap on a Circle problem* by Lee and Wu [4]. Given an instance of the Minimum Gap on a Circle problem, we do the following reduction to the MCCA′ problem:

(1) For every input point (x_i, y_i), compute (x_{i+}, y_{i+}) and create a demand arc A_i with its two ends specified by (x_i, y_i) and (x_{i+}, y_{i+}), where x_{i+} and y_{i+} are the unique solutions to the following equations for x and y:

$$(x-x_i)^2+(y-y_i)^2=\varepsilon^2$$

$$x^2+y^2=1$$

$$y<y_i.$$

Literally, (x_{i+}, y_{i+}) is the point on the unit circle ε away from (x_i, y_i) and is the CW end of the arc A_i.

(2) Use algorithm MCCA′ to compute m, the minimum number of cuts required to intersect all demand arcs.

(3) If $m=n$, then answer YES; otherwise answer NO.

The above reduction solves the Minimum Gap on a Circle problem using $O(n)$ effort in addition to the time spent in algorithm MCCA′; thus the MCCA′ requires $\Omega(n \log n)$ time. We can further treat the intersection of the tangent of the unit circle at (x_i, y_i) and the tangent of the unit circle at (x_{i+}, y_{i+}) as a demand point p_i for the MNF problem. Thus it is obvious that any algorithm that solves the MNF problem can also be used to solve the Minimum Gap on a Circle problem with $O(n)$ additional effort. Hence the MNF problem also requires $\Omega(n \log n)$ time. □

THEOREM 4. *The* MPS *problem requires* $\Omega(n \log n)$ *time.*

Proof. It is not difficult to show that the Minimum Gap on a Circle problem is linearly reducible to the MPS problem. □

THEOREM 5. *The maximum independent set of n circular-arcs can be found in* $O(n)$ *time if the endpoints have been sorted.*

Proof. Let $S=\{A_1, A_2, \cdots, A_n\}$ denote the set of arcs whose endpoints have been sorted, and let $\mathscr{H} \subseteq S$ denote a set of maximum independent arcs, no two of which intersect. We show that $|\mathscr{H}|$ is either equal to $|\mathscr{C}|$ or $|\mathscr{C}|-1$, where $\mathscr{C}=\{c_1, c_2, \cdots, c_m\}$ denotes the minimum cut set of S obtained by algorithm MCCA, and $\mathscr{S}(c_i)=c_{i+1}$ for $i=1, 2, \cdots, m-1$. Let $c_{m+1}=\mathscr{S}(c_m)$.

Case 1. $c_{m+1}=c_1$. In this case, each cut point c_i corresponds to an arc and no two of them overlap. We have $|\mathscr{H}| \geqq |\mathscr{C}|$. If $|\mathscr{H}|>|\mathscr{C}|$, there would exist two arcs falling betweeen two consecutive cut points, which is impossible. The set of arcs corresponding to $\mathscr{C}$ is a maximum independent set.

Case 2. $c_{m+1} \neq c_1$. In this case c_{m+1} must fall between c_1 and c_2. Note that the arc that determines c_m overlaps the arc that determines c_1. Let $\mathscr{H}=\{h_1, h_2, \cdots, h_t\}$. As before, we have $m-1 \leqq t \leqq m$. We show in this case $t=m-1$. Let ch_i be the cut interval determined by arc h_i for all $h_i \in \mathscr{H}$. It is obvious that no arc in $\mathscr{H}$ can span two consecutive cut points of $\mathscr{C}$. If t were equal to m, then there would exist a one-to-one correspondence $\mathscr{F}$ between the set of cut intervals $\mathscr{CH}$ and $\mathscr{C}$, i.e., $\mathscr{F}(ch_i)=(c_i)$. Consider now the subsequence $\mathscr{Q}'=(\mathscr{S}(c_m), \mathscr{S}^2(c_m), \cdots)$ of the repeating sequence $\mathscr{Q}*=(c_1, \mathscr{S}(c_1), \mathscr{S}^2(c_1), \cdots, \mathscr{S}^{m-1}(c_1), \cdots, \mathscr{S}^q(c_1))$ such that $\mathscr{S}^{q+1}(c_1)=c_1$. As shown in Lemma 5 (see Fig. 5), $\mathscr{S}(c_m)$ must fall between c_1 and ch_1, $\mathscr{S}^2(c_m)$ must fall between c_2 and $ch_2, \cdots$, and $\mathscr{S}^m(c_m)$ must fall between c_m and ch_m. Now $\mathscr{S}^{m+1}(c_m)$ must fall between (c_{m+1}) and ch_1 which is "away" from c_1. That is, the sequence $\mathscr{Q}*$ cannot repeat itself, a contradiction. The set of arcs corresponding to $\{c_1, c_2, \cdots, c_{m-1}\}$ is a maximum independent set. □

4. Conclusion. We have presented an optimal $\Theta(n \log n)$ algorithm to solve the MCCA problem for n circular-arcs. Based on this we have derived efficient (also optimal) solutions to the MNF problem and the MPS problem. We have also shown

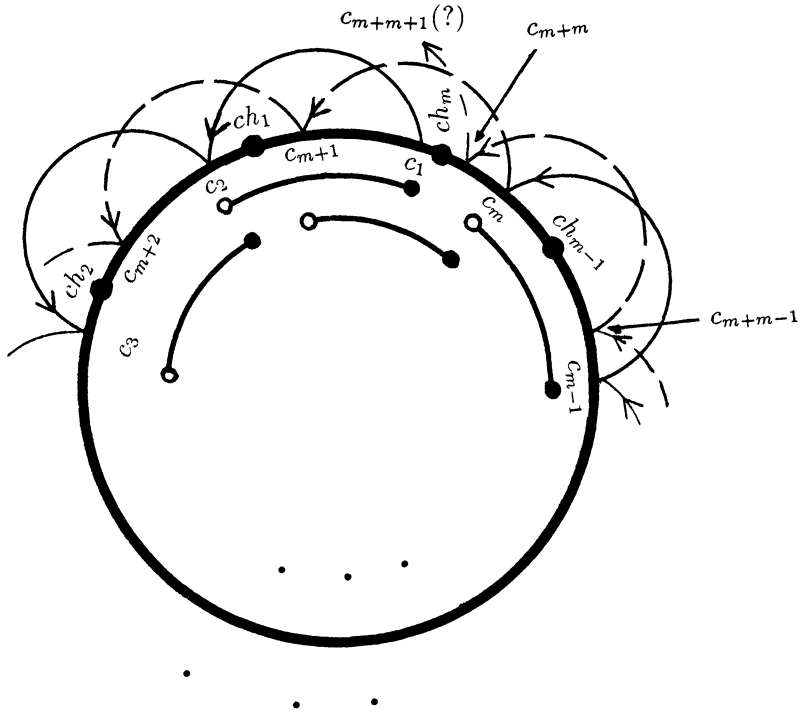


FIG. 5. *Proof of Theorem* 5.

that the maximum independent set of a set of n circular-arcs can be found in linear time if the endpoints of these arcs are sorted, a result much simpler than previously known results.

Acknowledgment. Concurrently and independently, Theorem 5 (i.e., a "simple" algorithm for finding a maximum independent set of n circular-arcs) was discovered by two other groups of researchers: (1) M. C. Golumbic and P. L. Hammer, *Stability in Circular Arc Graphs*, Journal of Algorithms, 9 (1988), pp. 314-320; and (2) W. L. Hsu and K. H. Tsai, *Linear Time Algorithm on Circular Arc Graphs*, 26th Annual Allerton Conference, September 1988, pp. 842-851.

REFERENCES

[1] M. BEN-OR, *Lower bounds for algebraic computation trees*, in Proc. 15th Annual Symposium on Theory of Computing, Boston, MA, 1983, pp. 80-86.

[2] Z. DREZNER, *Competitive location strategies for two facilities*, in Regional Science and Urban Economics 12, 1982, North-Holland, Amsterdam, pp. 485-493.

[3] S. L. HAKIMI, *On locating new facilities in a competitive environment*, in ISOLDE II, Skodsborg, Denmark, June 1981.

[4] D. T. LEE AND Y. F. WU, *Geometric complexity of some location problems*, Algorithmica, 1 (1985), pp. 193-211.

[5] S. MASUDA AND K. NAKAJIMA, *An optimal algorithm for finding a maximum independent set of a circular-arc graph*, SIAM J. Comput., 17 (1988), pp. 41-52.

[6] M. SARRAFZADEH AND D. T. LEE, *Topological via minimization revisited*, IEEE Trans. Comput., to appear.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1051–1058, December 1990

006

AN OPTIMAL $O(\log \log n)$ TIME PARALLEL STRING MATCHING ALGORITHM*

DANY BRESLAUER† AND ZVI GALIL‡

Abstract. An optimal $O(\log \log n)$ time parallel algorithm for string matching on CRCW-PRAM is presented. It improves previous results of Galil [*Inform. and Control*, 67 (1985), pp. 144–157] and Vishkin [*Inform. and Control*, 67 (1985), pp. 91–113].

Key words. string, period, parallel algorithms

AMS(MOS) subject classifications. 68Q10, 68Q20, 68Q25

1. Introduction. On a CRCW-PRAM we can solve some problems in less than the logarithmic time needed on weaker models such as CREW-PRAM. For example, *OR* and *AND* of n input variables, and finding the minimum or maximum of integers between 1 and n (see § 7) can be done in $O(1)$ time using n processors. Finding the maximum in the general case takes $O(\log \log n)$[1] time on $n/\log \log n$ processors ([14] and [13]), and the same is true for merging ([14], [9], and [3]). Recently, a few more $O(\log \log n)$ optimal parallel algorithms have been found for finding prefix minima [11], all nearest neighbors in convex polygons [12], triangulation of a monotone polygon, and nearest smallers [2]. We show that the string matching problem can be solved in $O(\log \log n)$ time with $n/\log \log n$ processors too, establishing that it belongs to one of the lowest parallel complexity classes.

The problem of string matching is defined as follows: Given two input arrays $TEXT(1 \cdots n)$ and $PATTERN(1 \cdots m)$, find all occurrences of the pattern in the text. Namely, find all indices j such that $TEXT(j+i-1) = PATTERN(i)$, for $i = 1 \cdots m$. In the sequential case, the problem can be solved, for example, using the two well-known linear time algorithms of Knuth, Morris, and Pratt [8] and Boyer and Moore [4]. In the parallel case, an optimal algorithm discovered by Galil [7] for fixed alphabet and later improved by Vishkin [15] for general alphabet solves the problem in $O(\log n)$ time on a CRCW-PRAM. Recall that an *optimal* parallel algorithm is one with a linear time-processor product. We use the weakest version of CRCW-PRAM: the only write conflict allowed is that processors can write the value 1 simultaneously into a memory location.

Our algorithm solves the string matching problem for general alphabet in $O(\log \log m)$ time using $n/\log \log m$ processors on a common CRCW-PRAM. It is based on the previous two optimal algorithms, and similarly works in two stages. In the first, we gather some information about the pattern and use it in the second stage to find all the occurrences of the pattern in the text. The output of the algorithm is a Boolean array $MATCH(1 \cdots n)$ that has the value "match" in each position where the pattern occurs and "unmatch" otherwise.

Suppose we have mn processors on a CRCW-PRAM. Then we can solve the string matching problem in $O(1)$ time using the following method:

- First, mark all possible occurrences of the pattern as "match."

* Received by the editors April 4, 1989; accepted for publication (in revised form) February 13, 1990. This work was supported by National Science Foundation grants CCR-86-05353 and CCR-88-14977.

† Columbia University, New York, New York 10027.

‡ Department of Computer Science, Columbia University, New York, New York 10027; and Tel-Aviv University, 69978 Tel-Aviv, Israel.

[1] All logarithms are to the base 2.

• To each such possible beginning of the pattern, assign m processors. Each processor compares one symbol of the pattern with the corresponding symbol of the text. If a mismatch is encountered, it marks the appropriate beginning as "unmatch."

Assuming that we can eliminate all but l of the possible occurrences (ignoring the problem of assigning the processors to their tasks), we can use the method described above to get an $O(1)$ parallel algorithm with lm processors. Both [7] and [15] use this approach. The only problem is that one can have many occcurrences of the pattern in the text, even much more than the n/m needed for optimality in the discussion above.

To eliminate this problem, we use the notion of the period suggested in [7] and also used in [15]. A string u is called a *period* of a string w if w is a prefix of u^k for some positive integer k or equivalently if w is a prefix of uw. We call the shortest period of a string w *the period* of w.

Suppose u is the period of the pattern w. As explained below, we cannot have two occurrences of the pattern at positions i and j of the text for $|j-i|<|u|$. If instead of matching the whole pattern, we look only for occurrences of u, assuming we could eliminate many of them and have only $n/|u|$ possible occurrences left, we can use the $O(1)$ algorithm described above to verify these occurrences using only n processors. Then by counting the number of consecutive matches of u, we can match the whole pattern.

In many cases, we slow down some computations to fit in our processor bounds. This is done using a theorem of Brent [5], which allows us to count only the number of operations performed without concern about their timing.

THEOREM (Brent). *Any synchronous parallel algorithm of time t that consists of a total of x elementary operations can be implemented on p processors in $\lceil x/p \rceil + t$ time.*

Using this theorem, for example, we can slow down the $O(1)$ time string matching algorithm described above to run in $O(s)$ time on lm/s processors.

Brent's theorem, as well as other computations described below, requires the assignment of processors to their tasks, which in our case is done using standard techniques.

In § 2 we review two facts on periods from [7] and in § 3 we review the notion of witness from [15]. In §§ 4–6 we describe the algorithm. Section 7 is devoted to some technicalities left out in the previous sections.

2. Periodicity properties. We will use some simple facts about periods in the next sections. The proof can be found in [7].

(1) If w has two periods of length p and q and $|w| \geq p+q$, then w has a period of length $\gcd(p, q)$ ([10]).

(2) If w occurs in positions p and q of some string and $0<q-p<|w|$, then w has a period of length $q-p$. Therefore we cannot have two occurrences of the pattern at positions p and q if $0<q-p<|u|$ and u is the period of the pattern.

3. Witnesses. An important idea in our algorithm is a method suggested in [15], which enables us to eliminate many possible occurrences in $O(1)$ time. One computes some information about the pattern, which is called $WITNESS(1 \cdots m)$ in [15], and uses it in the second stage for the analysis of the text.

Let u be the period of the pattern w, and let v be a prefix of w. It follows immediately from the periodicity properties that if $|u|$ does not divide $|v|$ and $|v|<\max(|u|, |w|-|u|)$, then v is not a period, and hence w is not a prefix of vw. In that case we can find an index k such that

$$PATTERN(k) \neq PATTERN(k-|v|).$$

We call this k a *witness to the mismatch of w and vw*, and define

$$WITNESS(|v|+1)=k.$$

We are interested only in $WITNESS(i)$ for $1<i\leqq|u|$, which by the periodicity properties mentioned above can be based only on the first $2|u|-1$ symbols of the pattern (If some $WITNESS(i)$ is greater than $2|u|$, it can be modified to be in the desired range: Let $r=WITNESS(i) \bmod |u|$; then if $r<i$ set $WITNESS\ (i)$ to $r+|u|$; otherwise set $WITNESS(i)$ to r).

4. Duels and counting. Assume that u is the period of the pattern w, $w=u^k v$, v is a proper prefix (possibly empty) of u, and $p=|u|$. We call the pattern *periodic* if its length is at least twice its period length (i.e., $m\geqq 2p$). Having computed the *WITNESS* array in the first stage, Vishkin [15] suggests the following method to eliminate close possible occurrences which he calls a *duel.* Suppose we suspect that the pattern may start at positions i and j of the text where $0<j-i<|u|$, thus, since we computed $r=WITNESS(j-i+1)$ we can find in $O(1)$ time a symbol in the text that will eliminate one or both of the possible occurrences. More specifically, since $PATTERN(r)\neq PATTERN(r-j+i)$, at most one of them can be equal to $TEXT(r+i-1)$ (see Fig. 1).

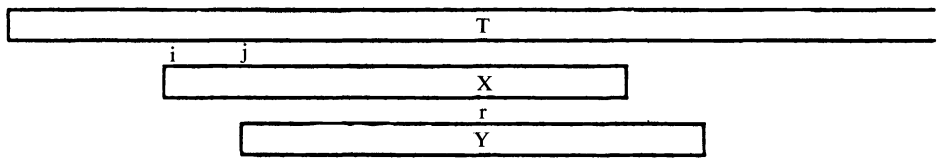


FIG. 1. *$X\neq Y$ and therefore we cannot have $T=X$ and $T=Y$.*

Actually, we eliminate possible occurrences of some prefix of the pattern. In the periodic case, we saw in the previous section that the witness information can be based only on the first $2p$ symbols of the pattern, thus we eliminate positions in which there is no occurrence of u^2. While in the nonperiodic case, the witness information is based on the whole pattern, and positions where there is no occurrence of it can be eliminated. Having many such duels in pairs, the algorithm of [15] eliminates enough possible occurrences of u in the text in $O(\log m)$ time and verifies them using the $O(1)$ time algorithm described above. We manage to reduce the time of [15] to $O(\log\log m)$ time algorithm using the following observations:

- Duels "work like" maximum. Having a block of the text of length equal to p, only one occurrence of the pattern might start in it. Assume that the pattern can start anywhere within that block, and suppose we have p^2 processors. Assign a processor to each pair and perform a duel. Since in every pair at least one loses, at the end we are left with no more than one possible occurrence in each block. The exact details of the algorithm appear in the next sections.
- We simplify the "counting" of consecutive occurrences of u in the text in the periodic case. A recent result of Beame and Hastad [1] shows that computing the parity of n bits on a CRCW-PRAM takes $\log n/\log\log n$ with any polynomial number of processors, so no "real" counting is possible within our time bounds. Assume without loss of generality that the text is of length $n=2m-p$ (divide the text into $n/(m-p)=O(n/m)$ overlapping groups of length $2m-p$). We call an occurrence of u^2 at position i an *initial occurrence* if there is no occurrence of it at position $i-p$. We call such an occurrence a *final occurrence* if there is no occurrence at position $i+p$.

The main observation is that there is at most one initial occurrence of interest which is the rightmost initial occurrence in the first $m-p$ positions. Any initial occurrence in a position greater than $m-p$ is of no interest, since there are not enough symbols in the text to match the whole pattern. Since the pattern is periodic with period length p, initial occurrences that are smaller cannot start occurrences of the pattern either. The correspondng final occurrence is the smallest final occurrence that is greater than the initial occurrence.

5. Processing the text. As we mentioned above, duels are like maxima. We describe an optimal $O(\log\log m)$ time text analysis based on having $WITNESS(2\cdots r)$, for $r=\min(p, \lceil m/2\rceil)$, computed in the pattern analysis stage that works similarly to the maximum finding algorithm of [13]. Recall that $p=|u|$ is the length of the period of the pattern. In the periodic case we divide the text into groups of length $n=2m-p$, whereas in the nonperiodic case we work on the whole text.

We have $WITNESS(i)<2p$. Partition the text into blocks of length r. We have n/r such blocks. In each block mark all positions as possible occurrences. Partition them into groups of size $\sqrt{r}$ and repeat recursively. The recursion bottoms out with one processor per block of size 1, where nothing is done. When done, we are left with one possible occurrence (or none) in each block of size $\sqrt{r}$, thus $\sqrt{r}$ possible occurrences altogether. Then in $O(1)$ time make all duels as described above. We are left with a single possible occurrence (or none) in each block of size r.

The algorithm described above takes $O(\log\log m)$ time but is not optimal; it requires n processors. To achieve optimality we first partition our block into small blocks of size $\log\log r$. To each one of the $r/\log\log r$ small blocks assign a processor and make duels between pairs using a sequential algorithm until left with at most one possible occurrence in each small block. Then, proceed with the $O(\log\log r)$ algorithm having at most $r/\log\log r$ possible occurrences to start with. Since we have n/r blocks and in each block we used $r/\log\log r$ processors, we need a total of $n/\log\log r$ processors for this computation. Left with at most n/r possible occurrences, we can use the $O(1)$ algorithm we described in the introduction to verify these occurrences. The next step depends on the periodicity of the pattern; we have two cases:

(1) The pattern is not periodic ($m<2p$, $r=m/2$): Verify the whole pattern at each possible occurrence. This can be done using $mn/r=2n$ processors in $O(1)$ time.

(2) The pattern is periodic:

- Verify at each possible occurrence in the text only the first $2p$ symbols of the pattern. This can be done using only $2n$ processors in $O(1)$ time.
- Find the initial occurrence and the corresponding final occurrence: First find all initial occurrences and final occurrences. Then , find the maximal initial occurrence in the first $m-p$ symbols and the corresponding final occurrence. This can be done $O(1)$ time using m processors on our weak CRCW-PRAM (see § 7).
- Verify v right after the final occurrence. Note that v occurs after each nonfinal occurrence since v is a prefix of u.
- For each verified occurrence of u^2 check to see if enough occurrences follow and whether or not they are followed by a verified occurrence of v. This can be done using the position of the initial occurrence and the final occurrence, and the information about v computed in the previous step.

Both (1) and (2) can be done in $O(1)$ time using n processors or $O(\log\log m)$ time using $n/\log\log m$ processors.

6. Processing the pattern. The *WITNESS* array that we used in the text processing stage is computed incrementally. Knowing that some witnesses were already computed

in previous stages, we can compute more witnesses easily. Let i and j be two indices in the pattern such that $i<j<\lceil m/2\rceil+1$. If $s = WITNESS(j-i+1)$ is already computed, then we can find at least one of $WITNESS(i)$ or $WITNESS(j)$ using a duel on the pattern as follows:

- If $s+i-1 \leqq m$, then $s+i-1$ is also a witness either for i or for j.
- If $s+i-1 > m$, then either s is a witness for j or $s-j+i$ is a witness for i (see Fig. 2).

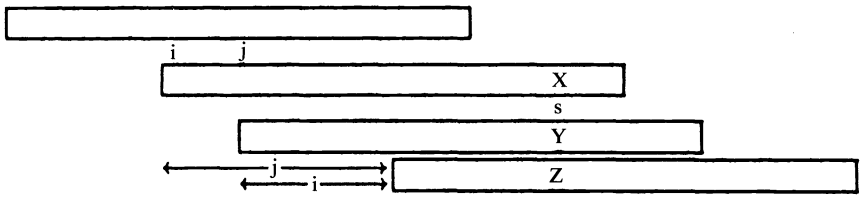


FIG. 2. *$X \neq Y$ and therefore we cannot have $Z = X$ and $Z = Y$.*

First we describe an $O(\log\log m)$ nonoptimal algorithm. It works in stages and it has at most $\log\log m$ stages. Let $k_i = m^{1-2^{-i}}$, $k_0 = 1$. At the end of stage i, we have at most one yet-to-be-computed witness in each block of size k_i. The only yet-to-be-computed index in the first block is 1.

(1) At the beginning of stage i we have at most k_i/k_{i-1} yet-to-be-computed witnesses in the first k_i-block. Try to compute them using the naive algorithm on $PATTERN(1\cdots 2k_i)$ only. This takes $O(1)$ time using $2k_i(k_i/k_{i-1}) = 2m$ processors.

(2) If we succeed in producing witnesses for all the indices in the first block (all but the first for which there is no witness), then we compute witnesses in each following block of the same size using the optimal duel algorithm described in the text processing section. This takes $O(\log\log m)$ time only for the first stage. In the following stages, we will have at most $\sqrt{m}$ indices for which we have no witness, and duels can be done in $O(1)$ time.

(3) If we fail to produce a witness for some $2 \leqq j \leqq k_i$, it follows that $PATTERN(1\cdots 2k_i)$ is periodic with period length p, where $p = j-1$ and j is the smallest index of a yet-to-be-computed witness. By the periodicity properties mentioned above, all yet-to-be-computed indices within the first block are of the form $kp+1$. Check periodicity with period length p to the end of the pattern. If p turns out to be the length of the period of the pattern, the pattern analysis is done and we can proceeed with the text analysis. Otherwise, the smallest witness found is good also for all the indices of the form $kp+1$ that are in the first k_i-block, and we can proceed with the duels as in (2).

These three steps seem to require a simultaneous write of different values. In the next section we show that our weaker CRCW-PRAM can do it too. In order to make our algorithm optimal, we take a more careful look at the algorithm described above. We redefine our block sizes k_i as follows,

$$k_0 = 1,$$

$$k_i = \frac{m^{1-2^{-i}}}{\log\log m}, \quad \text{for } i = 1\cdots\log\log m,$$

$$k_i = 2k_{i-1}, \quad \text{for } i > \log\log m,$$

introducing log log log m more stages. Using this new sequence, $m/\log\log m$ processors are enough for step 1 of the original algorithm. Step 2 will now take log log m time for the first **two** stages, after which we will have fewer than $(m/\log\log m)^{1/2}$ yet-to-be-computed witnesses. However, step 3 still needs m processors and we need to modify the entire algorithm.

We have two kinds of stages: nonperiodic stages and periodic stages. Each kind is associated with certain initial conditions. The first stage is a nonperiodic stage 1 for which the initial conditions hold vacuously because $k_0 = 1$ and no witnesses are computed.

A nonperiodic stage i starts with at most one yet-to-be-computed witness in each k_{i-1}-block (in the first k_{i-1}-block the yet-to-be-computed witness is always the first). Moreover, all computed witnesses satisfy

$$WITNESS(l) \leqq l + k_{i+1}. \tag{1}$$

A periodic stage i starts with some yet-to-be-computed witnesses in the first k_{i-1}-block. They are all the indices of the form $kp+1$, where p is the period length of the first k_i-block. In a periodic stage i all computed witnesses satisfy

$$WITNESS(l) \leqq l + k_i \tag{2}$$

and also,

$$WITNESS(l) \leqq 2p \leqq k_i \quad \text{for } 2 \leqq l \leqq p. \tag{3}$$

In a nonperiodic stage i we execute step 1 of our original algorithm, and if all witnesses in the first k_i-block are computed, we perform the duels of step 2, which result in at most one yet-to-be-computed witness in any k_i-block. The new witnesses in the first k_i-block obviously satisfy $WITNESS(l) \leqq 2k_i \leqq k_{i+1}$. Hence, the new witnesses in the other k_i-blocks satisfy $WITNESS(l) < l + k_{i+2}$. So all computed witnesses satisfy (1) with i increased by 1. If all witnesses in the first k_i-block have been computed, we proceed in a nonperiodic stage $i+1$; otherwise, we verify p to be the period length of the first k_{i+1}-block. If it is not, we find **the same** witness ($\leqq k_{i+1}$) for all the indices of the form $kp+1$ in the first k_i-block, and we continue with the duels of step 2 as in the previous case; otherwise, we proceed with a periodic stage $i+1$. In both cases, the initial conditions obviously hold.

In a periodic stage i we first check if p is the period length of the first k_{i+1}-block. In case it is, we use the periodicity to compute witnesses for all incides l, where $l \not\equiv 1 \pmod p$, in the first k_i-block as follows. Let $j = \lfloor (l-1)/p \rfloor p$. Set $WITNESS(l) = j + WITNESS(l-j) \leqq 2k_i \leqq k_{i+1}$ (by (3)). We then proceed with a periodic stage $i+1$, and the initial conditions obviously hold. Actually, (3) might not hold immediately. By (2) we have $WITNESS(l) < k_{i+1}$ for $2 \leqq l \leqq p$. Since p is the period length of the first k_{i+1}-block, we can modify the witnesses to satisfy (3) as in § 3.

If we find that p is not the period length of the first k_{i+1}-block, we actually find at once a witness for all indices of the form $kp+1$ in the first k_{i-1}-block. This witness is not larger than k_{i+1}. We then perform the duels in each of the k_{i-1}-blocks, which result in all computed witnesses satisfying (1) and with at most one yet-to-be-computed witness in each k_{i-1}-block. These are the initial conditions for a nonperiodic stage i. We then proceed with a nonperiodic stage i. Note that unlike the nonoptimal algorithm, we perform duels only if the next stage is nonperiodic.

We now take a careful look at the last stage. Let r be a maximal index such that $k_r < m$ and define $k_{r+1} = m$. As we have shown, duels can be made for all i and j where $i < j < \lceil m/2 \rceil + 1$; thus in a nonperiodic stage r everything works well if we perform

duels only in the first half of the pattern. In a periodic stage r we either verify the period of the whole pattern, or we find a witness and enter a nonperiodic stage r.

Since we can be in a periodic stage i and a nonperiodic stage i at most once for each i, the total number of operations is $O(m)$, and by Brent's theorem our algorithm is optimal.

7. Some detail. Our computation model is a CRCW-PRAM where the only write conflict allowed is that processors can write the value 1 simultaneously into a memory location. The duels of our text analysis can obviously be implemented on such a model, whereas the duels of the pattern analysis and a few other steps seem to require a stronger model of computation. We show how to implement the algorithm on our weaker model.

Consider the following problem: given an array of k integers, find the first 0. Fich, Ragde, and Wigderson [6] proposed the following $O(1)$ time algorithm using k processors on our weak CRCW-PRAM. Partition the array into $\sqrt{k}$ blocks of size $\sqrt{k}$. For each block find in $O(1)$ time if it has a 0 using $\sqrt{k}$ processors. Using $O(1)$ time minimum algorithm, find the first block that has a 0, and then, using the same algorithm, find in that particular block the first position of a 0.

Using this algorithm, we find the initial occurrence, the final occurrence, and witnesses in the first block in any stage of the pattern analysis without increasing our time/processor bounds on our weak CRCW-PRAM. The implementation of finding the initial occurrence, the final occurrence, and witnesses is obvious. However, the duels of the pattern analysis need to be done carefully. Using h^2 processors, suppose we perform duels among h indices. Each processor will write to a **different** memory location; then assign h processors to each of the h indices and check to see if a witness was found using the algorithm mentioned above.

We left out the details of the processor allocation for the duels, since it can be done exactly as in Shiloach and Vishkin's [13] maximum finding algorithm. We need to calculate some sizes for our algorithm and for the usage of Brent's theorem (i.e., k_i's). $\lfloor \log \log m \rfloor$ can be calculated in $O(\log \log m)$ time using a single processor, and square roots can be computed in $O(1)$ time on few processors as in [13].

As in [7], the text analysis can also be done in $O(\log 1/\varepsilon)$ time using nm^{ε} processors and the pattern analysis can be done in $O(1/\varepsilon)$ time using $m^{1+\varepsilon}$ processors.

REFERENCES

[1] P. BEAME AND J. HASTAD (1987), *Optimal bound for decision problems on the* CREW PRAM, in Proc. 19th ACM Symposium on Theory of Computing, New York, pp. 83–93.

[2] O. BERKMAN, B. SCHIEBER, AND U. VISHKIN (1988), *Some doubly logarithmic optimal parallel algorithms based on finding nearest smallers*, preprint.

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

[4] R. S. BOYER AND J. S. MOORE (1977), *A fast string searching algorithm*, Comm. ACM, 20, pp. 762–772.

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

[6] F. E. FICH, R. L. RAGDE, AND A. WIGDERSON (1984), *Relations between concurrent-write models of parallel computation*, in Proc. 3rd ACM Symposium on Principles of Distributed Computing, Vancouver, B.C., Canada, pp. 179–189.

[7] Z. GALIL (1985), *Optimal parallel algorithms for string matching*, Inform. and Control, 67, pp. 144–157.

[8] D. E. KNUTH, J. H. MORRIS, AND V. R. PRATT (1977), *Fast pattern matching in strings*, SIAM J. Comput., 6, pp. 322–350.

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

[10] R. C. LYNDON AND M. P. SCHUTZENBERGER (1962), *The equation $a^M = b^N c^P$ in a free group*, Michigan Math. J., 9, pp. 289–298.
[11] B. SCHIEBER (1987), *Design and analysis of some parallel algorithms*, Ph.D. thesis, Computer Science Department, Tel-Aviv University, Tel Aviv, Israel.
[12] B. SCHIEBER AND U. VISHKIN (1987), *The parallel complexity of finding all nearest neighbors in convex polygons*, preprint.
[13] Y. SHILOACH AND U. VISHKIN (1981), *Finding the maximum, merging and sorting in a parallel computation model*, J. Algorithms 2, pp. 88–102.
[14] L. G. VALIANT (1975), *Parallelism in comparison models*, SIAM J. Comput., 4, pp. 348–355.
[15] U. VISHKIN (1985), *Optimal parallel pattern matching in strings*, Inform. and Control, 67, pp. 91–113.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1059–1063, December 1990

007

FAST PARALLEL ALGORITHMS FOR SPARSE MULTIVARIATE POLYNOMIAL INTERPOLATION OVER FINITE FIELDS*

DIMA YU. GRIGORIEV†, MAREK KARPINSKI‡, AND MICHAEL F. SINGER§

Abstract. The authors consider the problem of reconstructing (i.e., interpolating) a t-sparse multivariate polynomial given a *black box* which will produce the value of the polynomial for any value of the arguments. It is shown that, if the polynomial has coefficients in a finite field $GF[q]$ and the black box can evaluate the polynomial in the field $GF[q^{\lceil 2\log_q(nt)+3\rceil}]$, where n is the number of variables, then there is an algorithm to interpolate the polynomial in $O(\log^3(nt))$ boolean parallel time and $O(n^2t^6\log^2 nt)$ processors.

This algorithm yields the first efficient deterministic polynomial time algorithm (and moreover boolean NC-algorithm) for interpolating t-sparse polynomials over finite fields and should be contrasted with the fact that efficient interpolation using a black box that only evaluates the polynomial at points in $GF[q]$ is not possible (cf. [M. Clausen, A. Dress, J. Grabmeier, and M. Karpinski, *Theoret. Comput. Sci.*, 1990, to appear]). This algorithm, together with the efficient deterministic interpolation algorithms for fields of characteristic 0 (cf. [D. Yu. Grigoriev and M. Karpinski, in *Proceedings of the 28th IEEE Symposium on the Foundations of Computer Science*, 1987, pp. 166–172], [M. Ben-Or and P. Tiwari, in *Proceedings of the 20th ACM Symposium on the Theory of Computing*, 1988, pp. 301–309]), yields for the first time the general deterministic sparse conversion algorithm working over arbitrary fields. (The reason for this is that every field of positive characteristic contains a primitive subfield of this characteristic, and so this method can be applied to the slight extension of this subfield.) The method of solution involves the polynomial enumeration techniques of [D. Yu. Grigoriev and M. Karpinski, *op. cit.*] combined with introducing a new general method of solving the problem of determining if a t-sparse polynomial is identical to zero by evaluating it in a *slight* extension of the coefficient field (i.e., an extension whose degree over this field is logarithmic in nt).

Key words. sparse multivariate polynomials, finite fields, interpolation

AMS(MOS) subject classifications. 68C25, 12C05

1. Introduction.

The polynomial interpolation algorithms play an important role in the design of efficient algorithms in algebra and their applications (cf. [G83], [G84], [K85], [BT88]). For the case of finite fields there were no deterministic polynomial time algorithms known (cf. [BT88]) for the sparse interpolation problem. The existing methods required large extension fields of order $GF[q^n]$; so, for example, no effective procedures for finding primitive elements over an actual interpolation field were known without using randomization.

Here we remedy the situation by considering what we call a "slight" extension of fields, which is an extension whose degree over the coefficient field is logarithmic in nt, $GF[q^{\lceil c\log_q(nt)\rceil}]$. The method of solution involves two major steps: (1) solving the zero identity problem of polynomials from $GF[q]$ by evaluating in a slight extension $GF[q^{\lceil 2\log_q(nt)+3\rceil}]$, and (2) using inductive enumeration of partial solutions for terms and coefficients over $GF[q]$ by means of recursion on (1). We develop a general method involving Cauchy matrices to solve the zero-identity problem in Step 1, and combine this with the refined polynomial enumeration techniques of Grigoriev and Karpinski [GK87] to solve Step 2.

Because of the lower bound of $\Omega(n^{\log t})$ (cf. [CDGK88]) for the interpolation over the same field $GF[q]$ without an extension, our *slight* field extension is in a sense the smallest extension capable of carrying out the efficient interpolation.

* Received by the editors March 17, 1989; accepted for publication (in revised form) March 4, 1990.

† Steklov Institute of Mathematics, Soviet Academy of Sciences, Leningrad 191011, USSR.

‡ Department of Computer Science, University of Bonn, 5300 Bonn 1, Federal Republic of Germany. The work of this author was supported in part by the Leibniz Center for Research in Computer Science and the Deutsche Forschungsgemeinschaft grant KA673/2-1.

§ Department of Mathematics, North Carolina State University, Raleigh, North Carolina 27695.

In what follows we shall use the basic notions of the theory of finite fields (cf. [LN86], [MS77]) and algorithms for computing in finite fields (cf. [L82]), and the basic models of parallel computation (cf. [C85], [G82]).

2. Interpolation problem over finite fields. We consider the problem of interpolation for multivariate polynomials given by *black boxes* (special cases of it are the explicit interpolations of polynomials given by straight-line programs (cf. [K85]), or polynomials given by determinants (cf. [L79], [GK87]). In this setting we are given a polynomial f in $GF[q]$ as a black box that allows us to evaluate f in extensions of $GF[q]$ and information about its sparsity t (the bound on the number of its nonzero coefficients). Given this, we must determine an extension $GF[q^s]$ of $GF[q]$, s as small as possible, and an efficient polynomial time interpolation algorithm working over $GF[q^s]$ to determine all coefficients of f in $GF[q]$.

We say that the *black box* interpolation problem (over a finite field extension $GF[q^S]$) is in NC^k (cf. [C85]), if there exists a class of uniform $(ntq)^{0(1)}$-size and $O(\log^k(ntq))$-depth boolean circuits with oracle nodes S (*returning* values of a black box over the field extension $GF[q^S]$) computing for an arbitrary n-variate polynomial $f \in GF[q][x_1, \cdots, x_n]$ all the nonzero coefficients and monomial vectors of f, with the oracle S_f^S, defined by $S_f^S(x_1, \cdots, x_n, y)$ if and only if $f(x_1, \cdots, x_n) = y$ over $GF[q^S]$. If the *lifting* of a *black box* (given explicitly by a straight-line program, determinant, boolean circuit, etc.) from $GF[q]$ to the extension $GF[q^S]$, and the computation of $f(x_1, \cdots, x_n)$ over $GF[q^S]$ by a black box, are both in boolean NC (in P), then the explicit interpolation problem lies also in boolean NC (in P).

We note that the interpolation problem over finite fields deals not only with the interpolation of polynomials but with arbitrary functions in their t-sparse ring sum expansion representation (RSE) ([W87]).

We shall develop an interpolation algorithm (for polynomials over $GF[q]$) for the *slight* extension of a field of order $s = \lceil 2\log(nt) + 3 \rceil$. This allows us for the first time to efficiently find the generators in $GF[q^S]$, as the size of this field is polynomial in the size of the input polynomial under interpolation. Our *slight* field extension is in a sense the best possible, as the efficient interpolation over the same field (i.e., for $s = 1$) is not possible. In [CDGK88] the tight lower and upper bounds $\Theta(n^{\log t})$ have been established for the number of steps needed to determine identity to zero of polynomials $f \in GF[2][x_1, \cdots, x_n]$.

3. The algorithm. We now formulate the Interpolation Theorem and the underlying Interpolation Algorithm over Finite Fields.

THE INTERPOLATION THEOREM. *Given any t-sparse polynomial $f \in GF[q][x_1, \cdots, x_n]$. For an arbitrary q, there exists a deterministic parallel algorithm (NC^3) for interpolating f over a slight field extension $GF[q^{\lceil 2\log_q(nt)+3 \rceil}]$ working in $O(\log^3(ntq))$ parallel boolean time and $O(n^2t^6\log^2(ntq) + q^{2.5}\log^2 q)$ processors. For a fixed field the algorithm works in $O(\log^3(nt))$ parallel boolean time and $O(n^2t^6\log^2 nt)$ processors.*

SPARSE INTERPOLATION ALGORITHM OVER FINITE FIELDS

Input: A black-box oracle allowing one to evaluate a t-sparse polynomial $f \in GF[q^s][x_1, \cdots, x_n]$ for $s = 1, \cdots$. (A t-sparse polynomial is a polynomial with at most t nonzero coefficients.)

Output: All $(\mathbf{k}, f_{\mathbf{k}})$ such that $f = \sum f_{\mathbf{k}} x^{\mathbf{k}}$ where $0 \neq f_{\mathbf{k}} \in GF[q]$ and $x^{\mathbf{k}} = x_1^{k_1} \cdot \ldots \cdot x_n^{k_n}$.

We begin by first describing a Subalgorithm.

SUBALGORITHM (IDENTITY-TO-ZERO TEST):

Input: Same as above.

Output: **Yes**, if $f \equiv 0$; **No**, if $f \not\equiv 0$.

Step 1: Choose s so that $q^s - 1 > 4nq(n-1)\binom{t}{2}$. So let $s = \lceil 2 \log_q (nt) + 3 \rceil$.

Step 2: Construct the field $GF[q^s]$ by looking over all polynomials of degree s with coefficients in $GF[q]$ and testing irreducibility with the help of the Berlekamp algorithm [B70]. We find an irreducible $\phi \in GF[q][z]$, and then $GF[q^s]$ is isomorphic to $GF[q][z]/(\phi)$. We find an ω that is generator of the cyclic group $GF[q^s]^*$ in the following way. Factor $q^s - 1 = \prod p_i^{n_i}$, p_i prime. For any $a \in GF[q^s]$, calculate $a^{(q^s-1)/p_i}$ for each i. We do this using the binary expansion of the exponent and by techniques from [L82]. An element is a generator of the cyclic group if and only if all these powers are distinct from 1.

Step 3: Denote $N = (\lceil q^s - 1 \rceil)/4nq$. Use the sieve of Eratosthenes to find a prime p with $2N < p \leqq 4N$. Such a prime exists by Bertrand's postulate (cf. [HW78]).

Step 4: Now construct an $N \times N$ Cauchy matrix C (cf. [C], [PS64], [MS77]) over the field $GF[p]$, $y_i = x_i = i$, $1 \leqq i \leqq N$ by $C = [1/(x_i + y_j)] = [1/(i+j)]$. We have

$$\det C = \frac{\prod_{1 \leqq i < j \leqq n} (x_j - x_i)(y_j - y_i)}{\prod_{1 \leqq i,j \leqq n} (x_i + y_i)}.$$

For any of its minors $\neq 0$, a similar formula holds. Therefore any minor of any size is nonsingular. Compute, using the Euclidean algorithm $c_{ij} \in \mathbb{Z}$, such that $c_{ij} \equiv 1/(i+j) (\mathrm{mod}\ p)$ and $0 \leqq c_{ij} < p \leqq 4N$.

Step 5: Denote by $\bar{C} = [\bar{c}_{ij}]$ an arbitrary submatrix of C of size $N \times n$.

Step 6: Pick out in parallel any row $\bar{c}_i = (\bar{c}_{ij})$, $1 \leqq j \leqq n$, of the matrix $\bar{C}$ and, for each l, $0 \leqq l < t$, plug $\omega^{l\bar{c}_{ij}}$ for each x_j in the black-box (with $s = \lceil 2 \log_q (nt) + 3 \rceil$) for the polynomial $f = \sum f_{\mathbf{k}} x^{\mathbf{k}} = \sum f_{\mathbf{k}} x_1^{k_1} \cdots x_n^{k_n}$, where $\mathbf{k} = (k_1, \cdots, k_n)$ and the number of $\mathbf{k}$'s is less than t, $0 \leqq k_j < q-1$, $f_{\mathbf{k}} \in GF[q]$.

We now pause to justify that if $f \not\equiv 0$, then for some $\bar{c}_{ij} l$ as above $f(\omega^{l\bar{c}_i}) \neq 0$, where $\omega^{l\bar{c}_{ij}}$ has been substituted for x_j. We first show that for a suitable vector $\bar{c}_i$, $1 \leqq i \leqq N$, after substituting $\omega^{\bar{c}_{ij}}$ for x_j, any two monomials $x^{\mathbf{k}}$, $x^{\mathbf{k}'}$ would give different elements of $GF[q]$. Suppose that for some pair $\mathbf{k}$, $\mathbf{k}'$ and $\bar{c}_i$ we have $\omega^{\bar{c}_i \cdot \mathbf{k}} = \omega^{\bar{c}_i \cdot \mathbf{k}'}$. This means that $\sum k_j \bar{c}_{ij} \equiv \sum k'_j \bar{c}_{ij} (\mathrm{mod}\ q^s - 1)$ and so $\sum (k_j - k'_j)\bar{c}_{ij} \equiv 0 (\mathrm{mod}\ q^s - 1)$. Since $|k_j - k'_j| \leqq q-1$, $\bar{c}_{ij} < 4N$, we have $\left|\sum_{1 \leqq j \leqq n} (k_j - k'_j)\bar{c}_{ij}\right| < (q-1)n4N < (q^s - 1)$; therefore $\sum (k_j - k'_j)c_{ij} = 0$. For any pair of monomials $x^{\mathbf{k}}$, $x^{\mathbf{k}'}$, we consider all the "bad" vectors $\bar{c}_i$, $1 \leqq i \leqq N$, i.e., those $\bar{c}_i$ for which $\sum_{1 \leqq j \leqq n} (k_j - k')\bar{c}_{ij} = 0$. There cannot be more than $(n-1)$ "bad" vectors for this pair, since if there exist such n vectors $\bar{c}_{i_1}, \cdots, \bar{c}_{i_n}$, the corresponding $n \times n$ submatrix of $\bar{C}$ would have determinant zero. As there are at most $\binom{t}{2}$ pairs of monomials, there is a vector $\bar{c}_{i_0}$, $1 \leqq i_0 \leqq N$, that is not "bad" for any pair of monomials $\mathbf{k}$, $\mathbf{k}'$, since $\binom{t}{2}(n-1) < N$.

Let $\bar{c}_{i_0}$ be some vector such that distinct monomials $x^{\mathbf{k}}$, $x^{\mathbf{k}'}$ yield distinct elements of $GF[q^s]$ after substituting $\omega^{\bar{c}_{i_0}}$. We now show that $f(\omega^{l\bar{c}_{i_0}}) \neq 0$ for some $0 \leqq l < t$. If $f(\omega^{l\bar{c}_{i_0}}) = 0$ for all l, $0 \leqq l < t$, then $XV = 0$, where $X = (f_{\mathbf{k}})_{\mathbf{k}}$ and $V = (\omega^{l\bar{c}_{i_0} \cdot \mathbf{k}})$ is the $t \times t$ matrix whose rows are indexed by l, $0 \leqq l < t$, and columns are indexed by the $\mathbf{k}$ that appears as an exponent in f.

Note that $\det(V)^2 = \prod_{\mathbf{k} \neq \mathbf{k}'} (\omega^{\sum \bar{c}_{j i_0} j^{k_j}} - \omega^{\sum \bar{c}_{j i_0} j^{k_j}}) \neq 0$ (it is a Vandermonde matrix), so we have a contradiction. Therefore the identity-to-zero subalgorithm is correct.

We now continue with the main algorithm. Assume $n = 2^m$ for simplicity of notation. Define $S_{\alpha,\beta} = \{(k_1, \cdots, k_{2^{\alpha-1}}) : x_{\beta 2^{\alpha-1}+1}^{k_1} \cdot \ldots \cdot x_{\beta 2^{\alpha-1}+2^{\alpha-1}}^{k_{2^{\alpha-1}}}$ occurs as a subterm in some nonzero term of $f\}$, where $1 \leqq \alpha \leqq m+1$ and $0 \leqq \beta < 2^{m+1-\alpha}$. We produce $S_{\alpha,\beta}$ recursively for $\alpha = 1, \cdots, m+1$.

Basis Step: Let $\alpha = 1$. Let $\{a_1, a_2, \cdots\}$ be an enumeration of $GF[q]$. In parallel for each $a \in GF[q]$, substitute a for $x_{\beta+1}$ in f. Find a vector $u_l \in (GF[q])^q$ such that $u_l \cdot (a_i^j) = (0, \cdots, 1, \cdots, 0)$ where all entries of this latter vector are 0 except for a 1 in the lth place. We then have $u_l \cdot (f(x_1, \cdots, x_\beta, a_1, x_{\beta+2}, \cdots, x_n), \cdots, f(x_1, \cdots, x_\beta, a_q, x_{\beta+2}, \cdots, x_n)) = P_l$ where $f = \sum_l x_{\beta+1}^l P_l$ and $P_l \in GF[q][x_1, \cdots, x_\beta, x_{\beta+2}, \cdots, x_n]$. We see that P_l may be evaluated at any point $(b_1, \cdots, b_{\beta-1}, b_{\beta+1}, \cdots, b_n)$ by evaluating f at the q points $(b_1, \cdots, b_{\beta-1}, a_i, b_{\beta+1}, \cdots, b_n)$, $i = 1, \cdots, q$ and using this last formula, where u_l has been found by inverting the matrix (a_i^j) and extracting the lth row. This gives a black box for P_l. The identity-to-zero subalgorithm now allows us to determine which P_l's are not identically zero, and so to determine $S_{1,\beta}$.

Recursion Step: Assume that we have produced $S_{\alpha,\beta}$ for all β, $0 \leqq \beta < 2^{m+1-\alpha}$. We now produce $S_{\alpha+1,\beta}$ for fixed β, $0 \leqq \beta < 2^{m-\alpha}$. For each element from the set $S_{\alpha,2\beta}$ and for each element from the set $S_{\alpha,2\beta+1}$, consider the corresponding product $x_{\beta 2^\alpha+1}^k, \cdots, x_{\beta 2^\alpha+2^\alpha}^{k_{2^\alpha}}$. For all such products (observe that the number of them is at most t^2, since $|S_{\alpha,2\beta}|, |S_{\alpha,2\beta+1}| \leqq t$), we can find (in parallel) a vector $v \in \mathbb{N}^{2^\alpha}$ as in Step 6 such that $v = (v_1, \cdots, v_{2^\alpha})$, $0 \leqq v_i < 4N_1$, where s_1 is chosen such that $(\ulcorner q^{s_1} - 1 \urcorner)/4nq = N_1 > (n-1)\binom{t^2}{2}$ and for any two products $x_{\beta 2^\alpha+1}^{k_1} \cdot \ldots \cdot x_{\beta 2^\alpha+2^\alpha}^{k_{2^\alpha}}$ and $x_{\beta 2^\alpha+1}^{k'_1} \cdot \ldots \cdot x_{\beta 2^\alpha+2^\alpha}^{k'_{2^\alpha}}$, $q^{s_1} - 1 \nmid (\sum k_i v_i - \sum k'_1 v_i)$. Let $\omega_1 \in GF[q^{s_1}]$ be a generator of the cyclic group $GF[q^{s_1}]^*$. For any $0 \leqq l < t^2$, we replace $x_{\beta 2^\alpha+j}$ with $\omega_1^{v_j l}$. Consider the $t^2 \times t^2$ matrix $B = (\omega_1^{(\sum_j k_j v_j) l}) = (b_{\mathbf{k}}, l)$. Note that $\det(B)^2 = \prod_{\mathbf{k} \neq \mathbf{k}'} (\omega_1^{(\sum_j k_j v_j)} - \omega_1^{(\sum_j k'_j v_j)}) \neq 0$, since $q^{s_1} - 1 \nmid (\sum_j k_j v_j - \sum k'_j v_j)$. Calculate vectors $u_j \in (GF[q^{s_1}])^{t^2}$ such that $u_j B = (0, \cdots, 0, 1, 0, \cdots, 0)$ where this latter vector has 1 in the ith position and zeros everywhere else. We then have $u_i \cdot Y = \bar{P}_i$ where $f = \sum_{\mathbf{k}} x^{\mathbf{k}} \bar{P}_{\mathbf{k}}$, where $x^{\mathbf{k}} = x_{\beta 2^\alpha+1}^{k_1} \cdot \ldots \cdot x_{\beta 2^\alpha+2^\alpha}^{k_{2^\alpha}}$ and $\bar{P}_{\mathbf{k}} \in GF[q][x_1, \cdots, x_{\beta 2^\alpha}, x_{(\beta+1)2^\alpha+1}, \cdots, x_n]$, and Y is the $1 \times t^2$ vector whose lth entry is $f(x_1, \cdots, x_{\beta 2^\alpha}, \omega_1^{v_1 l}, \cdots, \omega_1^{v_{2^\alpha} l}, x_{(\beta+1)2^\alpha+1}, \cdots, x_n)$. Using this last formula with black box evaluations of f gives us the new black boxes for the $\overline{P_i}$ as before. The *identity-to-zero* subalgorithm now allows us to determine which $\overline{P_i}$ are not identically zero and thus to determine $S_{\alpha+1,\beta}$. Notice that when $\alpha = m+1$ we have determined all the terms of f in the form of $(\mathbf{k}, f_{\mathbf{k}})$ such that $f = \sum_{\mathbf{k}} f_{\mathbf{k}} x^{\mathbf{k}}$, $0 \neq f_{\mathbf{k}} \in F[q]$ and $x^{\mathbf{k}} = x_1^{k_1}, \cdots, x_n^{k_n}$. □

4. Analysis of the Algorithm. Let $N = (\ulcorner q^{s-1} \urcorner / 4nq)$. Note that $N < nt^2 q$. The parallel time of our algorithm is $O(\log^3 N)$. This is because the *identity-to-zero* test takes $O(\log^2 N)$ parallel time, the recursive step calls this test and uses matrix inversion, which requires $O(\log^2 N)$ parallel time [M86], and the recursion depth is $O(\log n)$. Steps 1-5 take $O(N \log^2 (Nnq))$ processors. Step 6 takes $O(Nnt \log^2 (Nnq))$ processors. Therefore the total cost (in processors) of the identity-to-zero subalgorithm is $O(Nnt \log^2 (Nnq))$.

We now proceed to analyze the complexity of the rest of the algorithm. In the basic step, we must invert the $q \times q$ matrix (a_i^j) over $GF[q]$. This requires $O(q^{2.5} \log^2 q)$ processors by [M86]. In applying Steps 1-6 to test whether P_l is identically zero, we refer q times to substituting $\omega^{\bar{c}_{ij}}$ in a black box and calling the *identity-to-zero* test. Thus we need $Nntq \log^2 Nnq$ processors. In the recursion step, we calculate $N_1 t^2$ sums $\sum_j k_j v_j$ of length n and compute $\omega_1^{\sum_j k_j v_j}$ in the field $GF[q^{s_1}]$. This takes $N_1 t^2 n \log^2 N_1$ processors. Notice that $N_1 < nt^4 q$. Inverting the $t^2 \times t^2$ matrix B over $GF[q^{s_1}]$ requires $t^5 \log^2 N_1$ processors [M86]. Therefore the total number of processors would be $O(t^6 n^2 q \log^2 (tnq) + q^{2.5} \log^2 q)$. For a fixed field, the algorithm works in $O(\log^3 nt)$ time and $O(n^2 t^6 \log^2 nt)$ processors.

5. Further research. Our parallel algorithm enjoys very good parallel time bound. Concerning the number of processors, would it be possible to improve on the number of processors of the interpolation algorithm?

Acknowledgments. We are grateful to Michael Ben-Or, Johannes Grabmeier, Michael Rabin, Volker Strassen, and Avi Wigderson for a number of interesting conversations.

REFERENCES

[AL86] L. M. ADLEMAN AND H. K. LENSTRA, *Finding irreducible polynomials over finite fields*, in Proceedings of the 18th ACM Symposium on the Theory of Computing, 1986, pp. 350–355.

[B70] E. R. BERLEKAMP, *Factoring polynomials over large finite fields*, Math. Comp., 24 (1970), pp. 713–735.

[B81] M. BEN-OR, *Probabilistic algorithms in finite fields*, in Proceedings of the 22nd IEEE Symposium on the Foundations of Computer Science, 1981, pp. 394–398.

[BT88] M. BEN-OR AND P. TIWARI, *A deterministic algorithm for sparse multivariate polynomial interpolation*, in Proceedings of the 20th ACM Symposium on the Theory of Computing, 1988, pp. 301–309.

[C] A. L. CAUCHY, *Exercises d'analyse et de physics mathematiques*, Vol. 2, Bachelier, Paris, 1841, pp. 151–159.

[C85] S. A. COOK, *A taxonomy of problems with fast parallel algorithms*, Inform. and Control, 64 (1985), pp. 2–22.

[CDGK] M. CLAUSEN, A. DRESS, J. GRABMEIER, AND M. KARPINSKI, *On zero-testing and interpolation of k-sparse multivariate polynomials over finite fields*, Theoret. Comput. Sci., 1990, to appear.

[G82] L. GOLDSCHLAGER, *Synchronous parallel computation*, J. Assoc. Comput. Mach., 29 (1982), pp. 1073–1086.

[G83] J. VON ZUR GATHEN, *Factoring sparse multivariate polynomials*, in Proceedings of the 24th IEEE Symposium on the Foundations of Computer Science, 1983, pp. 172–179.

[G84] ———, *Parallel algorithms for algebraic problems*, SIAM J. Comput., 13 (1984), pp. 808–824.

[GK87] D. YU. GRIGORIEV AND M. KARPINSKI, *The matching problem for bipartite graphs with polynomially bounded permanents is in* NC, in Proceedings of the 28th IEEE Symposium on the Foundations on Computer Science, 1987, pp. 166–172.

[HW78] G. H. HARDY AND E. M. WRIGHT, *An Introduction to the Theory of Numbers*, Fifth Edition, Oxford University Press, London, 1978.

[K85] E. KALTOFEN, *Computing with polynomials given by straight-line programs in greatest common divisors*, in Proceedings of the 17th ACM Symposium on the Theory of Computing, 1985, pp. 131–142.

[L79] L. LOVÀSZ, *On determinants, matchings, and random algorithms*, in Fundamentals of Computation Theory, Akademie-Verlag, Berlin, 1979, pp. 565–574.

[L82] R. LOOS, *Computing in algebraic extensions*, in Computer Algebra: Symbolic and Algebraic Computation, Springer-Verlag, Berlin, New York, 1982, pp. 173–187.

[L83] A. K. LENSTRA, *Factoring multivariate polynomials over finite fields*, in Proceedings of the 15th ACM Symposium on the Theory of Computing, 1983, pp. 189–192.

[LN86] H. LIDL AND H. NIEDERREITER, *Introduction to Finite Fields and Their Applications*, Cambridge University Press, London, 1986.

[MS72] F. J. MACWILLIAMS AND N. J. A. SLOANE, *The Theory of Error-Correcting Codes*, North-Holland, Amsterdam, 1977.

[M86] K. MULMULEY, *A fast parallel algorithm to compute the rank of a matrix over an arbitrary field*, in Proceedings of the 18th ACM Symposium on the Theory of Computing, 1986, pp. 338–339.

[PS64] G. PÒLYA AND G. SZEGÖ, *Aufgaben und Lehrsätze aus der Analysis*, Vol. 2, Springer-Verlag, Berlin, New York, 1964.

[S80] J. T. SCHWARTZ, *Fast probabilistic algorithms for verification of polynomial identities*, J. Assoc. Comput. Mach., 27 (1980), pp. 701–717.

[W87] L. WEGENER, *The Complexity of Boolean Functions*, John Wiley, New York, 1987.

[Z79] R. E. ZIPPEL, *Probabilistic algorithms for sparse polynomials*, in Proc. EUROSAM '79, Springer Lecture Notes in Computer Science, 72 (1979), pp. 216–226.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1064-1067, December 1990

008

LINEAR CIRCUITS OVER GF(2)*

NOGA ALON†, MAURICIO KARCHMER‡, AND AVI WIGDERSON§

Abstract. For $n = 2^k$, let S be an $n \times n$ matrix whose rows and columns are indexed by $\mathrm{GF}(2)^k$ and, for $i, j \in \mathrm{GF}(2)^k$, $S_{i,j} = \langle i, j \rangle$, the standard inner product. Size-depth trade-offs are investigated for computing $S\mathbf{x}$ with circuits using only linear operations. In particular, linear size circuits with depth bounded by the inverse of an Ackerman function are constructed, and it is shown that depth two circuits require $\Omega(n \log n)$ size. The lower bound applies to any Hadamard matrix.

Key words. size-depth trade-offs, Boolean circuits, linear circuits, graph covers, Hadamard matrices

AMS(MOS) subject classifications. 05B20, 68R05, 94C15

1. Introduction. Let F be a field, and A a fixed $n \times n$ matrix with entries in F. There are no nontrivial lower bounds for computing the linear transformation $A\mathbf{x}$ where $\mathbf{x} \in F^n$ is an input, even if the circuit uses only linear operations.

When F is GF(2), linear operations are not more than exclusive-or gates. Counting arguments show that for a random matrix A, circuits of size $\Omega(n^2/\log n)$ are needed. In fact, $O(n^2/\log n)$ is an upper bound on the size needed for every matrix [B]. However, no explicit matrix A is known which requires superlinear size, even if the depth is restricted to be $O(\log n)$. (Valiant [V] has given an algebraic condition on matrices that would imply such a lower bound, but no matrix satisfying this condition has been constructed.)

In this note we consider H, the Boolean Hadamard matrix, and investigate size-depth trade-offs for computing $H\mathbf{x}$.

2. Definitions. A *Boolean Hadamard matrix* H is a matrix with entries in GF(2) and such that every two rows have Hamming distance $n/2$. Note that a Boolean Hadamard matrix can be constructed from a Hadamard matrix H' by $H = \frac{1}{2}(J + H')$ where J is the matrix of all ones. For $n = 2^k$, the *Sylvester Boolean matrix*, S, is one whose rows and columns are indexed by $\mathrm{GF}(2)^k$ and, for $i, j \in \mathrm{GF}(2)^k$, $S_{i,j} = \langle i, j \rangle$, the inner product of i and j. It is easy to show that S is a Boolean Hadamard matrix.

A *circuit* for $y = B\mathbf{x}$ where B is an $m \times n$ Boolean matrix is a DAG with n input nodes $x_1, \cdots, x_n$, m output nodes $y_1, \cdots, y_m$ and every noninput node computing the sum mod 2 of its inputs. (There is no bound on the fanin or on the fanout.) The *size* of the circuit is its number of edges. The *depth* is the length of the longest directed input–output path. Let $s(B)$ denote the size of the smallest circuit for $B\mathbf{x}$, and let $s_d(B)$ be the smallest size when the depth of the circuit is restricted to d.

The following lemma is important in understanding size-depth trade-offs.

LEMMA 2.1. *Let A, B be any two Boolean matrices. Then:*

(1) $s(B) = s(B^T)$, *where B^T is the transpose of B.*

(2) $s(AB) \leqq s(A) + s(B)$ *if A and B can be multiplied together.*

(3) $s(B) = s(B^\pi)$, *where B^π is the matrix B with rows permuted according to π.*

Furthermore, the same is true for depth restricted circuits where (2) *is replaced by* $s_{d_1+d_2}(AB) \leqq s_{d_1}(A) + s_{d_2}(B)$.

* Received by the editors February 14, 1989; accepted for publication (in revised form) April 18, 1990.
† Institute of Mathematics and Computer Science, Tel-Aviv University, Tel Aviv, Israel.
‡ Department of Mathematics, Massachusetts Institute of Technology, Cambridge, Massachusetts 02139.
§ Institute of Mathematics and Computer Science, Hebrew University, Jerusalem 91904, Israel.

Proof. (1) Let $\mathscr{C}$ be a circuit for $B\mathbf{x}$ with input nodes $\mathscr{I}=\{I_1, \cdots, I_n\}$ and output nodes $\mathscr{O}=\{O_1, \cdots, O_m\}$. Note that I_i has an odd number of paths to O_j if and only if $B_{i,j}=1$. Hence, by switching the roles of $\mathscr{I}$ and $\mathscr{O}$ and reversing all edges in $\mathscr{C}$, we get a circuit for $B^T\mathbf{x}$.

(2) Let $\mathscr{C}_A(\mathscr{C}_B)$ be a circuit for $A\mathbf{x}$ ($B\mathbf{x}$) with input nodes $\mathscr{I}_A$ ($\mathscr{I}_B$) and output nodes $\mathscr{O}_A$ ($\mathscr{O}_B$). It is easy to see that the circuit obtained by identifying $\mathscr{O}_A$ with $\mathscr{I}_B$ computes $AB\mathbf{x}$.

(3) Let $\mathscr{C}$ be a circuit for $B\mathbf{x}$ with input nodes $\mathscr{I}=\{I_1, \cdots, I_n\}$. By permuting $\mathscr{I}$ according to π and redirecting the edges going out of $\mathscr{I}$ we get a circuit for $B^{\pi}\mathbf{x}$.

The depth restricted claims can be proved similarly. □

3. Known results. The following results are known (see [B]):

THEOREM 3.1 [B]. *For most $n\times n$ Boolean matrices B, $s(B)=\Omega(n^2/\log n)$.*

THEOREM 3.2 [B]. *For every $n\times n$ Boolean matrix B, $s_2(B)=O(n^2/\log n)$.*

FACT 3.1. *For every $n\times n$ Boolean matrix B, $s_1(B)=\omega(B)$, where $\omega(B)$ is the number of ones in B.*

The only specific matrix that has been studied in some detail is the parallel prefix matrix P, where $P_{i,j}=1$ if and only if $i\leqq j$. We will first define some very slowly growing functions as in [CFL]. Let $A(0,j)=2j$; $A(i,1)=2$; and $A(i,j)=A(i-1, A(i,j-1))$ be the Ackerman function. Let $\alpha(n,d)=\min\{j: A(d,j)\geqq n\}$. Furthermore, let $\alpha(n)=\min\{j: A(j,j)\geqq n\}$.

THEOREM 3.3 [CFL]. $s_d(P)=O(n\alpha(n,d))$.

In particular, this theorem implies the following corollaries.

COROLLARY 3.1. $s_{O(1)}(P)=O(n\alpha(n))$.

COROLLARY 3.2. $s_{\alpha(n)}(P)=O(n)$.

4. New results. We present the following results.

THEOREM 4.1. $s_{2d}(S)=O(n\alpha(n,d))$ *where S is the Sylvester Boolean matrix.*

In particular, S can be computed in linear size and $\alpha(n)$ depth. This seems different from the behaviour of similar matrices (say FFT) over other fields which are conjectured to have size $\Omega(n\log n)$ regardless of the depth.

Though we would dare to conjecture that the above bounds are the best possible, we can only prove them for $d=2$, namely, Theorem 4.2.

THEOREM 4.2. *If H is a Boolean Hadamard matrix, then $s_2(H)=\Omega(n\log n)$.*

As far as we know, this is the first nontrivial lower bound for these circuits, even in the restricted case of depth two. Another interpretation of Theorem 4.2 has to do with a nonmonotone version of the problem of covering a graph with complete bipartite graphs. The monotone question was studied in connection to lower bounds on monotone, depth-three formulae (Hansel, Krichevskii [S]). In the same vein, the combinatorial question below relates to lower bounds on nonmonotone, depth-three formulae. Let $G=([n],[n],E)$ be a bipartite graph.[1] Let $\mathscr{K}$ be a collection of complete bipartite graphs $A_i\times B_i$, where $A_i, B_i\subseteq[n]$. We say that $\mathscr{K}$ covers G if $(i,j)\in E$ if and only if (i,j) appears in an odd number of graphs in $\mathscr{K}$. Let $|\mathscr{K}|=\sum_i(|A_i|+|B_i|)$ and define the cover number of G, $\beta(G)$ as the minimum of $|\mathscr{K}|$, where $\mathscr{K}$ covers G.

A Boolean matrix B can be considered as the adjacency matrix of a bipartite graph G_B. Given a depth-two circuit $\mathscr{C}$ for B, and a node v in the middle layer, we can associate a complete bipartite graph $A(v)\times B(v)$ where $A(v)$ ($B(v)$) is the set of nodes with edges to (from) v. With this as a hint, the proof of the following fact is left for the reader.

[1] $[n]=\{1, \cdots, n\}$.

FACT 4.1. $s_2(B) = \beta(G_B)$.

We thus get the following corollary.

COROLLARY 4.1. $\beta(G_S) = \Theta(n \log n)$ *where* S *is the Sylvester Boolean matrix.*

It is an open problem to construct a graph G with $\beta(G) = \Omega(n^{1+\varepsilon})$.

5. Upper bounds. We now prove Theorem 4.1. Recall that we are working over GF (2). Let $n = 2^k$. Then, by definition, $S = DD^T$, where D is the $2^k \times k$ matrix whose rows are all the vectors in GF $(2)^k$. By Lemma 2.1, we have that $s(S) \leqq s(D) + s(D^T) = 2s(D)$.

A *Grey code* is an ordering $v_1, \cdots, v_{2^k}$ of GF $(2)^k$ such that $\omega(v_i \oplus v_{i+1}) = 1$ for all i (i.e., the Hamming distance of every two consecutive vectors is one). Once again, by Lemma 2.1 we can assume, without loss of generality, that the rows of D are $v_1, \cdots, v_{2^k}$ as above.

Let $u_1 = (0, 0, \cdots, 0)$, and let $u_i = v_{i-1} \oplus v_i$ for $2 \leqq i \leqq 2^k$. Let U be the matrix whose rows are $u_1, \cdots, u_{2^k}$. Clearly, $s(U) \leqq \omega(U) \leqq n - 1$. Furthermore, $D = PU$ where P is the parallel prefix matrix as defined in § 3. We thus get $s(S) \leqq 2(s(P) + n - 1)$ and, by Theorem 3.3, $s_{2d}(S) = O(n\alpha(n, d))$.

6. Lower bounds. We give two different lower bounds for $s_2(H)$. The first is weaker than Theorem 4.2, but uses only the combinatorial structure of Hadamard matrices. The proof of Theorem 4.2 will use the algebraic structure of Hadamard matrices, together with results of Valiant [V] and Alon and Maass [AM].

Let $\mathscr{C}$ be a circuit for H and let $\mathscr{I}$ and $\mathscr{O}$ be the set of inputs and outputs of $\mathscr{C}$, respectively. Furthermore, let $\mathscr{M}$ be the set of nodes of $\mathscr{C}$ in the middle layer. Without loss of generality, we may assume that all edges of $\mathscr{C}$ are either in $\mathscr{I} \times \mathscr{M}$ or in $\mathscr{M} \times \mathscr{O}$.

The following combinatorial fact will be needed. A *sunflower* with k petals is a set system $\{R_1, \cdots, R_k\}$, where $R_i = C \dot{\cup} Z_i$ and the Z_i's are pairwise disjoint. C is called the *center* of the sunflower and the Z_i's are called the *petals.* The following theorem is well known.

THEOREM 6.1 (Erdös-Rado). *Every family of* $r!\,k^r$ *sets each of which has cardinality less than* r *contains a sunflower with* k *petals.*

Now we prove the weak version of Theorem 4.2.

THEOREM 6.2. *For any Hadamard matrix* H, $s_2(H) = \Omega(n \log n / \log\log n)$.

Proof. Let E_1 be the set of edges of $\mathscr{C}$ in $\mathscr{I} \times \mathscr{M}$, and let E_2 be the set of edges in $\mathscr{M} \times \mathscr{O}$. Let $m = c \log n / \log\log n$, for some constant c to be determined later. We will show that if $|E_1| \leqq mn$ then $|E_2| \geqq mn/2$, proving the theorem.

Let $S \subseteq \mathscr{I}$ be the set of inputs with (out)degree at most $2m$. Clearly, $|S| \geqq n/2$. For a vertex $i \in S$, let $T_i \subseteq \mathscr{M}$ be its set of neighbours. The collection of T_i's for $i \in S$ forms a set system of many small sets. By the sunflower theorem, there exists $R \subseteq S$, $|R| = 2m$ such that $\{T_i \mid i \in R\}$ form a sunflower. Let C be the center of the sunflower and $\{Z_i \mid i \in R\}$ its petals. Let F_i denote the edges of E_2 emanating from Z_i. □

CLAIM 6.1. *For every* $i, j \in R$, $|F_i| + |F_j| \geqq n/2$.

The theorem follows from Claim 6.1 by pairing the elements of R into $\{(i_1, j_1), \cdots, (i_m, j_m)\}$ arbitrarily so that

$$|E_2| \geqq \left| \bigcup_{i \in R} F_i \right| = \sum_{i \in R} |F_i| = \sum_{l=1}^{m} (|F_{i_l}| + |F_{j_l}|) \geqq \frac{mn}{2}.$$

Proof of claim. Let $K \subseteq \mathscr{O}$ be the set of outputs that depend on exactly one of the inputs x_i and x_j. By the definition of H, $|K| = n/2$. For every element $k \in K$, the number of paths from i to k must have a different parity than the number of paths from j to

k. But then, there must exist at least one edge from $Z_i \cup Z_j$ to k. Since this is true for every $k \in K$, $|F_i| + |F_j| \geq n/2$. □

Now we prove Theorem 4.2. $S_2(H) = \Omega(n \log n)$. For $S \subseteq \mathcal{I}$ and $T \subseteq \mathcal{O}$, let $L(S, T) \subseteq \mathcal{M}$ be the nodes in $\mathcal{M}$ connected both to nodes in S and T. Let $H_{S,T}$ be the $|S| \times |T|$ submatrix of H indexed by S and T. The following observation of Valiant [V] is the key to our proof.

FACT 6.1. $|L(S, T)| \geq \text{rank}\,(H_{S,T})$.

Intuitively, this fact says that the information contained in linearly independent values cannot be compressed. We will use the following theorem of Alon and Maass [AM].

THEOREM 6.3. *If for every $S \subseteq \mathcal{I}$ and $T \subseteq \mathcal{O}$ with $|S| = |T| = n^{1/2+2\varepsilon}$, $|L(S, T)| \geq \varepsilon \log n$, then $\mathcal{C}$ has at least $\Omega(n \log n)$ edges.*

Hence it will suffice to prove Claim 6.2.

CLAIM 6.2. *Let $|S| = |T| = n^{1/2+2\varepsilon}$, then* $\text{rank}\,(H_{S,T}) \geq \varepsilon \log n$.

Proof. Assume this is not so. Then there are at most n^{ε} different columns in $H_{S,T}$ so that one appears at least $n^{1/2+\varepsilon}$ many times. Without loss of generality, assume that this column has more ones than zeros. Then H contains a monochromatic submatrix of size $(n^{1/2+\varepsilon}) \times (n^{1/2+\varepsilon}/2)$, which contradicts the well-known fact [L] that every monochromatic submatrix of H has area at most $4n$. □

REFERENCES

[AM] N. ALON AND W. MAASS, *Meanders, Ramsey theory and lower bounds for branching programs*, in Proc. 27th Annual IEEE Symposium on Foundations of Computer Science, IEEE Computer Society, Washington, DC, 1986, pp. 410–417.

[B] S. BUBLITZ, *Decomposition of graphs and monotone formula size of homogeneous functions*, Acta Inform., 23 (1986), pp. 689–696.

[CFL] A. K. CHANDRA, S. FORTUNE, AND R. LIPTON, *Unbounded fanin circuits and associative functions*, in Proc. 15th Annual ACM Symposium on Theory of Computing, Association for Computing Machinery, New York, 1983, pp. 52–60.

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

[S] J. SAVAGE, *The Complexity of Computing*, John Wiley, New York, 1976.

[V] L. VALIANT, *Graph-theoretic arguments in low-level complexity*, in Lecture Notes in Computer Science, Vol. 53, Springer-Verlag, Berlin, New York, 1977.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1068-1099, December 1990

009

RANDOM POLYNOMIALS AND APPROXIMATE ZEROS OF NEWTON'S METHOD*

JOEL FRIEDMAN†

Abstract. In this paper the authors study the size of the set of "approximate zeros" for Newton's method, for a randomly chosen polynomial over certain distributions. For a degree d monic polynomial with coefficients chosen uniformly and independently in the unit ball, the results of this paper show, for example, that the set of approximate zeros is at least $C d^{(-1.5-\varepsilon)}$ for any positive ε, with C depending only on ε.

Key words. Newton's method, random polynomials, approximate zeros

AMS(MOS) subject classifications. primary 30C15; secondary 65H05

1. Introduction. The goal of this paper is to give average case analyses of a randomized version of Newton's method. For a function $f: \mathbf{C} \to \mathbf{C}$ we define Newton's map

$$T_f(z) = z - \frac{f(z)}{f'(z)}.$$

$z \in \mathbf{C}$ is said to be an *approximate zero* of f (for Newton's method) if the iterates of z, $z_0 = z$, $z_1 = T_f(z_0)$, $z_2 = T_f(z_1), \cdots$ converge to a root of f, ζ, and converge quickly enough so that

$$|z_k - z_{k-1}| \leqq \left(\frac{1}{2}\right)^{2^{k-1}-1} |z_1 - z_0|.$$

It is easy to see that the above condition implies

$$|z_k - \zeta| \leqq \frac{7}{2}\left(\frac{1}{2}\right)^{2^{k-1}} |z_0 - \zeta|.$$

The following *α-test* was proven independently in [Kim85] and [Sma86a].

LEMMA 1.1. *For some constant $\alpha_0 > 0$, $\alpha(f, z) < \alpha_0$ implies that z is an approximate zero of f, where*

$$\alpha(f, z) \equiv \frac{|f(z)|}{|f'(z)|} \sup_{k>1} \left|\frac{f^{(k)}(z)}{k! f'(z)}\right|^{1/(k-1)}.$$

Following [Sma86b], we consider the following randomized version of Newton's method. For $f \in P_d(1) = \{f(z) = z^d + a_1 z^{d-1} + \cdots + a_d \,|\, |a_i| \leqq 1\}$ choose z with $|z| \leqq 3$ at random and see if $\alpha(f, z) < \alpha_0$. If not, repeat the random choice until we find a z with $\alpha(f, z) < \alpha_0$. Then apply Newton's method, which is known to converge very quickly, some small number of times. Since Newton's method converges quickly there, the main cost of the algorithm will be the number of times needed to pick z's until we

* Received by the editors November 30, 1987; accepted for publication (in revised form) March 6, 1990. Part of this work was done at University of California at Berkeley, and was supported by an IBM (Yorktown Heights) fellowship and an IBM (Almaden) R.S.A.

† Department of Computer Science, Princeton University, Princeton, New Jersey 08544.

find one with $\alpha(f, z) < \alpha_0$ (times the cost of verifying this condition). Let

$$\Lambda_f \equiv \{z \in \mathbf{C} \mid \alpha(f, z) < \alpha_0\},$$

and let

$$\lambda(f) \equiv \frac{|\Lambda_f \cap B_3(0)|}{|B_3(0)|},$$

the density of Λ_f in $B_3(0)$ with respect to Lebesgue's measure. If z is chosen uniformly in $B_3(0)$, then the expected time to a z with $\alpha(f, z) < \alpha_0$ is $1/\lambda(f)$.

Unfortunately, there exist polynomials in $P_d(1)$ with arbitrarily small $\lambda(f)$—in particular, Λ_f will be contained in a small ball about 0 if the roots of f are contained in a sufficiently small ball about 0. We therefore seek to estimate $\lambda(f)$ for a typical polynomial f.

View $P_d(1)$ as a probability space with uniform distribution as a bounded subset of $\mathbf{C}^d$. In [Sma86a] Smale proves that the expected value of $\lambda(f)$ over $f \in P_d(1)$ is at least c/d^5 for some positive constant c. To be more specific, let $Q(\varepsilon)$ be the set of polynomials in $P_d(1)$ with $\lambda(f) < \varepsilon$. Smale obtained the estimate

$$\Pr\{Q(\varepsilon)\} < cd^5\varepsilon \tag{1.1}$$

for some absolute constant c.

In this paper we use a different approach to estimate $\Pr\{Q(\varepsilon)\}$, which gives estimates for various distributions of random polynomials. We obtain improved estimates, such as those given in the following theorem.

THEOREM 1.2. *For the uniform distribution on $P_d(1)$ we have for any integer N a c such that*

$$\Pr\{Q(\varepsilon)\} < c\left(\varepsilon^2 d^3 + \left(\varepsilon \log \frac{1}{\varepsilon}\right)^{(N-1)/2} d^N\right).$$

This shows that the expected value of $\lambda(f)$ is $> cd^{-2-\beta}$ for any $\beta > 0$, and that $\Pr\{Q(\varepsilon)\}$ decays like ε^2 rather than ε. For other versions, see Theorems 7.2, 4.12, 5.8, and 6.5. For polynomials with roots chosen independently and uniformly in $B_1(0)$ we get for any $\eta > 0$ constants c, c' such that

$$\Pr\{Q(\varepsilon)\} < c(c\varepsilon d)^{c' \min(d, (\varepsilon d)^{\eta - 1})}.$$

The term *approximate zero* first appeared in [Sma81]. There Smale defined a weaker notion of approximate zero (exponential as opposed to doubly exponential convergence) and proved that an iterate of 0 under a relaxation of Newton's method[1] is an approximate zero (with bounds on how large an iterate). Related papers include [SS85] and [SS86]. Before that, double exponential convergence of Newton's method was proven under conditions on the values of f and f' at a point and of f'' in a region; this was done by Kantorovich in [Kan52]; see also [KA70] and [GT74]. Independently, Kim in [Kim85] and Smale in [Sma86b] discovered the α-test. Kim used Schlicht function theory and obtained $\alpha_0 = 1/54$. Smale proved the α-test in the more general Banach space setting (e.g., Newton's method for maps: $\mathbf{C}^n \to \mathbf{C}^n$) and obtained $\alpha_0 = .1307\cdots$. Royden, in [Roy86], has recently improved the best known α_0 value to $.15767\cdots$ for maps $\mathbf{C} \to \mathbf{C}$.

[1] Namely, $T_{f,h} = z - h(f'(z)/f(z))$ with $0 < h < 1$.

Our method of proof obtains an estimate of $\Pr\{Q(\varepsilon)\}$ in terms of the distribution of the roots, which is proven in § 2.[2] In § 3 we apply this to the distribution on f where we take the roots to be chosen independently with uniform distribution.

In §§ 4–7 we estimate $\Pr\{Q(\varepsilon)\}$ for f with coefficients chosen independently. This leads us to the problem of determining the distribution of the roots given independently chosen coefficients. This problem has received a lot of attention (see [BS86]), but most of it is concentrated on estimating the density function of one randomly chosen root of the polynomial (i.e., "the condensed distribution"). We are interested in the joint density of two or more roots. To do this we use a generalized formula of Hammersley (see [Ham60]) for the joint density of two or more roots. In § 4 we calculate the joint density of two roots assuming the coefficients are distributed normally, and then prove a theorem about the density of approximate zeros. In § 5 we show that if the coefficients are distributed uniformly, similar results hold for the joint density of two roots and thus about the density of approximate zeros. These results also hold for a wider class of bounded distributions. In § 6 we refine the estimate of $\Pr\{Q(\varepsilon)\}$ given in §§ 4–5 by estimating the joint density of three or more roots. In § 7 we use an estimate of Erdös and Turán on the distribution of the roots to improve our $\Pr\{Q(\varepsilon)\}$ estimates further, and to obtain Theorem 1.2.

We mention that Coppersmith has pointed out that if the estimates of § 5 held for all values of the joint density function, we could prove that the expected value of $\lambda(f)$ over uniform $f \in P_d(1)$ is $> c/(d \log^2 d)$. This could well be the case, but the estimating of the joint density function is quite involved, and it is only in certain ranges of values that the joint density function is estimated. We also mention that a referee has pointed out that for ε smaller than roughly d^{-7}, $\Pr\{Q(\varepsilon)\}$ decays very quickly, more precisely $\Pr\{Q(\varepsilon)\} \leqq (c\varepsilon d^7)^{d/2}$ for some constant c.

2. Distances of roots.

LEMMA 2.1. *Let $x_1, \cdots, x_d$ be the roots of f. Let*

$$r = \frac{1}{\sum_{j>1} 1/|x_j - x_1|}.$$

For any constant $\alpha_0 > 0$ there is a constant $c > 0$ independent of f and r such that $\alpha(f, z) < \alpha_0$ for all $z \in B_{cr}(x_1)$.

Proof. Consider $g(y) = f(y - z) = \sum_{i=0}^{d} b_i y_i$, which has roots $x_i - z$. We wish to bound

$$\alpha(f, z) = \left|\frac{b_0}{b_1}\right| \max_{k>1} \left|\frac{b_k}{b_1}\right|^{1/(k-1)}.$$

Consider $h(y) = \sum_{i=0}^{d} b_{d-i} y^i$. Since $h(y) = y^d g(1/y)$, h has roots $1/(x_i - z)$ and thus

$$\sum_{i=0}^{d} b_{d-i} y^i = h(y) = b_0 \prod_{i=1}^{d} \left(y - \frac{1}{x_i - z}\right).$$

Thus

$$b_i = (-1)^i b_0 \sigma_i\left(\frac{1}{x_1 - z}, \cdots, \frac{1}{x_d - z}\right),$$

[2] Independently, Renegar (see [Ren87]) has discovered such an estimate, though his is weaker by a factor of anywhere from d^2 to d^4.

where σ_k is the kth symmetric polynomial,

$$\sigma_k(w_1, \cdots, w_d) = \sum_{i_1 < \cdots < i_k} w_{i_1} w_{i_2} \cdots w_{i_d}.$$

Now

$$\sigma_k\left(\frac{1}{x_1 - z}, \cdots, \frac{1}{x_d - z}\right) = \sigma_k\left(\frac{1}{x_2 - z}, \cdots, \frac{1}{x_d - z}\right) + \frac{1}{x_1 - z}\,\sigma_{k-1}\left(\frac{1}{x_2 - z}, \cdots, \frac{1}{x_d - z}\right).$$

Since $z \in B_{cr}(x_1)$ we have

$$\frac{c}{|x_1 - z|} > \sum_{j>1} \frac{1}{|x_j - z|}.$$

Thus

$$\left|\sigma_m\left(\frac{1}{x_2 - z}, \cdots, \frac{1}{x_d - z}\right)\right| \leqq \left(\sum_{i=2}^{d} \frac{1}{|x_i - z|}\right)^m \leqq \left(\frac{c}{|x_1 - z|}\right)^m$$

and

$$\begin{aligned}\left|\sigma_m\left(\frac{1}{x_1 - z}, \cdots, \frac{1}{x_d - z}\right)\right| &\leqq \left|\sigma_m\left(\frac{1}{x_2 - z}, \cdots, \frac{1}{x_d - z}\right)\right| \\ &\quad + \frac{1}{|x_1 - z|}\left|\sigma_{m-1}\left(\frac{1}{x_2 - z}, \cdots, \frac{1}{x_d - z}\right)\right| \\ &\leqq \frac{c^m + c^{m-1}}{|x_1 - z|^m}.\end{aligned}$$

On the other hand,

$$\begin{aligned}\left|\sigma_1\left(\frac{1}{x_1 - z}, \cdots, \frac{1}{x_d - z}\right)\right| &= \left|\frac{1}{x_1 - z} + \cdots + \frac{1}{x_d - z}\right| \\ &\geqq \frac{1}{|x_1 - z|} - \sum_{i=2}^{d} \frac{1}{|x_i - z|} \\ &\geqq \frac{1 - c}{|x_1 - z|}.\end{aligned}$$

Hence

$$\left|\frac{b_m}{b_1}\right| \leqq \frac{c^m + c^{m-1}}{1 - c} \frac{1}{|x_1 - z|^{m-1}}$$

and

$$\left|\frac{b_0}{b_1}\right| \leqq \frac{|x_1 - z|}{1 - c}.$$

Thus

$$\begin{aligned}\alpha(f, z) &\leqq \frac{|x_1 - z|}{1 - c} \max_{k>1} \left(\frac{c^k + c^{k-1}}{1 - c}\right)^{1/(k-1)} \frac{1}{|x_1 - z|} \\ &= \frac{1}{1 - c} \max_{k>1} c\left(\frac{1 + c}{1 - c}\right)^{1/(k-1)} = \frac{c}{1 - c}\sqrt{\frac{1 + c}{1 - c}},\end{aligned}$$

and hence $\alpha(f, z) < \alpha_0$ for the appropriate choice of c.

3. The uniform root distribution. We can use Lemma 2.1 to estimate the measure of $Q(\varepsilon)$ for various distributions on the set of degree d polynomials. In this section we illustrate this by carrying out such an estimate in a case where the roots are distributed independently. In this case we can apply Lemma 2.1 without much difficulty. Consider the distribution on polynomials

$$f(z)=(z-x_1)\cdots(z-x_d)$$

with $x_1,\cdots,x_d$ chosen independently, uniform in $B_1(0)\subset\mathbf{C}$. We begin by proving $\Pr\{Q(\varepsilon)\}\leqq c d\varepsilon$ for some c, and then we refine the argument to get $\Pr\{Q(\varepsilon)\}\leqq(cd\varepsilon)^d$ for some c.

THEOREM 3.1. $\Pr\{Q(\varepsilon)\}\leqq cd\varepsilon$ *for all* ε *for some absolute constant* c.

Proof. Viewing x_1 as fixed, we have for any fixed j

$$\Pr\{|x_j-x_1|<\rho\}=\frac{|B_\rho(0)\cap B_1(0)|}{|B_1(0)|}\leqq\rho^2.$$

Thus, for any $i_1<\cdots<i_k$, we have

$$\Pr\{|x_{i_1}-x_1|,\cdots,|x_{i_k}-x_1| \text{ are } \leqq\rho\}\leqq\rho^{2k}.$$

Hence

$$\Pr\{k \text{ of } |x_2-x_1|,|x_3-x_1|,\cdots,|x_d-x_1| \text{ are } \leqq\rho\}\leqq\binom{d-1}{k}\rho^{2k}\leqq\left(\frac{ed}{k}\right)^k\rho^{2k},$$

(where $\binom{d-1}{k}\leqq\binom{d}{k}\leqq(ed/k)^k$ was used), which is $\leqq\eta/2^k$ if $\rho\leqq(k/(2ed))^{1/2}\eta^{1/2k}$. So if $a_1\leqq\cdots\leqq a_{d-1}$ are the $|x_2-x_1|,\cdots,|x_d-x_1|$ arranged in increasing order, we have

$$\Pr\left\{a_1\leqq\sqrt{\frac{1}{2ed}}\,\eta^{1/2} \text{ or } a_2\leqq\sqrt{\frac{2}{2ed}}\,\eta^{1/4} \text{ or } \cdots \text{ or } a_{d-1}\leqq\sqrt{\frac{d-1}{2ed}}\,\eta^{1/2(d-1)}\right\}$$

$$\leqq\frac{\eta}{2}+\frac{\eta}{4}+\cdots+\frac{\eta}{2^{d-1}}<\eta.$$

Hence with probability $\geqq1-\eta$ we have

$$\sum_{i=1}^{d-1}\frac{1}{a_i}\leqq\sqrt{2ed}\left(\left(\frac{1}{\eta}\right)^{1/2}+\frac{1}{\sqrt{2}}\left(\frac{1}{\eta}\right)^{1/4}+\cdots+\frac{1}{\sqrt{d-1}}\left(\frac{1}{\eta}\right)^{1/2(d-1)}\right). \tag{3.1}$$

LEMMA 3.2. $N+(1/\sqrt{2})N^{1/2}+\cdots+(1/\sqrt{m})N^{1/m}\leqq2N+4\sqrt{m}$ *for* $N\geqq4$.

Proof. Let $t>1$ be the first integer for which $N^{1/t}\leqq2$. Then $t-1\leqq\log_2 N$ and so

$$N+\frac{1}{\sqrt{2}}N^{1/2}+\cdots+\frac{1}{\sqrt{t-1}}N^{1/(t-1)}\leqq N+(t-2)N^{1/2}$$

$$\leqq N+(\log_2 N)N^{1/2}\leqq2N,$$

since $\log_2 N\leqq N^{1/2}$ for $N\geqq4$. Furthermore,

$$\frac{1}{\sqrt{t}}N^{1/t}+\cdots+\frac{1}{\sqrt{m}}N^{1/m}\leqq2\left(\frac{1}{\sqrt{2}}+\cdots+\frac{1}{\sqrt{m}}\right)\leqq2\int_1^m\frac{1}{\sqrt{x}}\,dx\leqq4\sqrt{m}$$

and the lemma follows.

Applying Lemma 3.2 to (3.1) yields that with probability $\leqq1-\eta$,

$$\sum_{i=1}^{d-1}\frac{1}{a_i}\leqq\sqrt{2\pi d}\,(2\sqrt{1/\eta}+4\sqrt{d})$$

(for $\sqrt{1/\eta} \geqq 4$), and if $1/\eta \geqq d$, this gives

$$\sum_{i=1}^{d-1} \frac{1}{a_i} \leqq \sqrt{2\pi d}\,(2\sqrt{1/\eta} + 2\sqrt{1/\eta}) \leqq \sqrt{8\pi}\,\sqrt{d/\eta}$$

and hence

$$\frac{1}{\sum \frac{1}{a_i}} \geqq c\sqrt{\eta/d}$$

for some constant c. Hence, by Lemma 2.1, $\alpha(f, z) < \alpha_0$ in a ball about x_1 of area

$$c\frac{\eta}{d}.$$

Applying the arguments with x_1 replaced by an arbitrary root, it follows that with probability $\geqq 1 - d\eta$, each root has $\alpha(f, z) \leqq \alpha_0$ in a ball about it of area $c\eta/d$, for a total area of $c\eta$. □

Theorem 3.1 implies that the expected value of $\lambda(f)$ is $> c/d$. We now remark that the rate of decay of $\Pr\{Q(\varepsilon)\}$ can easily be improved upon, although we will not obtain a better estimate of the expected value of $\lambda(f)$.

To improve the decay rate, let m be a positive integer to be specified later, and superimpose a square grid over $B_1(0)$ with squares of side length $\sqrt{dm\pi}$, so that $B_1(0)$ is subdivided into $dm + O(\sqrt{dm})$ pieces, with all but $O(\sqrt{dm})$ of the pieces being whole squares. For appropriately chosen l and δ, we will estimate the probability that (1) no square contains more than l roots, (2) at least $d/2$ of the squares contain exactly one root, and (3) at least, say, $d/100$ of the squares in condition (2) have their roots being of distance at least $\delta/\sqrt{dm\pi}$ away from the boundary of the square. Assuming these three conditions hold, we get for each of the $d/100$ roots, x_i, of condition (3), that

$$\sum_{j \neq i} \frac{1}{|x_j - x_i|} < c\delta^{-1}\sqrt{dm\pi} + c \sum_{k=1}^{O(\sqrt{d/l})\sqrt{dm}} \frac{}{k}\, lk,$$

the sum over k representing squares that are k squares away (say, in the Manhatten distance) from the original square, and the above is

$$\leqq c\delta^{-1}\sqrt{dm} + cd\sqrt{ml}.$$

Thus the above three conditions guarantee $\lambda(f)$ to be at least

$$cd \min\left(\frac{\delta^2}{dml}, \frac{1}{d^2 ml}\right).$$

On the other hand, condition (1) does not hold with probability

$$\leqq dm \sum_{r \geqq l} \binom{d}{r}\left(\frac{1}{dm}\right)^r \left(1 - \frac{1}{dm}\right)^{d-r} \leqq dmc\binom{d}{l}\left(\frac{1}{dm}\right)^l \leqq cdm\left(\frac{1}{lm}\right)^l.$$

If condition (2) does not hold, then there must be some $d/4$ roots which land in squares occupied by the other $3d/4$ roots; this occurs with probability

$$\leqq \binom{d}{d/4}\left(\frac{3d/4}{dm}\right)^{d/4} \leqq \left(\frac{c}{m}\right)^{d/4}.$$

Finally if condition (2) does hold, condition (3) will fail to hold with probability

$$\leqq \sum_{r \leqq d/100} \binom{d/2}{r}(4\delta - 4\delta^2)^{d/2-r}(1 - 4\delta + 4\delta^2)^r,$$

which, for δ less than some absolute constant, is

$$\leqq (4\delta)^{d/4}.$$

Given ε, assuming that $\varepsilon < c/d$ for a small $c > 0$, we take $\delta = (d\varepsilon)^{1/5}$, $m = (d\varepsilon)^{-1/10}$, and $l = 1/(\varepsilon dm)$ yields

$$\Pr\{Q(\varepsilon)\} \leqq c(c\varepsilon d)^{c' \min(d, (\varepsilon d)^{-9/10})}$$

for constants c and c'. Similarly adjusting δ, m, and l we can replace the above 9/10 with any constant <1.

4. Normally distributed coefficients. In the next sections we will estimate $\Pr\{Q_d(\varepsilon)\}$ for distributions in which the coefficients a_i of

$$f(z) = a_d z^d + \cdots + a_0 \tag{4.1}$$

are chosen independently with fixed distributions. In this section we consider the case in which the a_i's are distributed normally, i.e., $\Re(a_i)$ and $\Im(a_i)$ are independent random variables on $\mathbf{R}$ with density

$$\phi(t) = \frac{1}{\sqrt{2\pi}} e^{-t^2/2}.$$

Here we have the problem that for any fixed value of d, there is some small probability that all the coefficients are large enough to enable $B_3(0)$ to lie completely within a sink of period 2. Hence $\Pr\{Q_d(0)\} > 0$ for each d; we cannot hope to prove $\Pr\{Q_d(\varepsilon)\} \leqq \varepsilon d^{O(1)}$. Instead, we shall prove

$$\Pr\{Q_d(\varepsilon)\} \leqq c_1 \varepsilon^2 d^7 + 2^{-c_2 d}, \tag{4.2}$$

the $2^{-c_2 d}$ term taking $\Pr\{Q_d(0)\} > 0$ into account.

We begin by noting that for d large the roots tend to be located on the circle of radius 1.

LEMMA 4.1 (Specht). *Let $z_1, \cdots, z_m$ be roots of $z^n + a_{n-1} z^{n-1} + \cdots + a_0$. Then*

$$|z_1 \cdots z_m| \leqq \sqrt{1 + |a_{d-1}|^2 + |a_{d-2}|^2 + \cdots + |a_0|^2}.$$

Proof. For the proof, see [Mar66]. □

COROLLARY 4.2. *If fewer than $d/2$ of the roots of $f(z) = a_d z^d + \cdots + a_0$ have absolute value between $\frac{1}{2}$ and 2, then either*

$$\frac{1}{|a_d|} \sqrt{|a_0|^2 + \cdots + |a_d|^2} \geqq 2^{d/4} \tag{4.3}$$

or

$$\frac{1}{|a_0|} \sqrt{|a_0|^2 + \cdots + |a_d|^2} \geqq \left(\frac{1}{2}\right)^{-d/4} = 2^{d/4}. \tag{4.4}$$

Proof. Either $d/4$ roots have absolute value >2 or $<\frac{1}{2}$. Apply Lemma 4.1 to either $f(z)/a_n$ or $f(1/z)z^d/a_0$.

COROLLARY 4.3. *With probability $\geqq 1 - 2^{cd}$ there are at least $d/2$ roots z in the range $\frac{1}{2} \leqq |z| \leqq 2$ for some constant $c > 0$.*

Proof. For (4.3) or (4.4) to hold, one of the a_i's must be exponentially large or exponentially small (i.e., $\geqq 2^{cd}$ or $\leqq 2^{-cd}$). For standard normal random variables, this occurs with probability $\leqq 2^{-cd}$ for some $c \geqq 0$.

Next we derive a bound of the form $\Pr\{|z_1 - z_2| \leqq \varepsilon\} \leqq O(\varepsilon^4)$ ($O(\varepsilon^4)$ for fixed d) where z_1, z_2 are randomly chosen roots of $f(z) = a_d z^d + \cdots + a_0$. Contrasting this to $\Pr\{|z_1 - z_2| \leqq \varepsilon\} = O(\varepsilon^2)$ when z_1, z_2 are distributed independently and uniformly explains why in (4.2) we estimate $\Pr\{Q(\varepsilon)\}$ quadratically in ε rather than linearly (i.e., (3.1)).

In [Ham60], Hammersley gives a formula for the density function,[3] $P(z_1)$, of a randomly chosen root, z_1, of $f(z) = a_d z^d + \cdots + a_0$. Viewing $f(z_1)$ and $f'(z_1)$, for z_1 fixed, as sums of independent random variables $(z_1^d)a_d + \cdots + a_0$ and $(dz_1^{d-1})a_d + \cdots + a_1$, the formula for P can be written as

$$P(z_1) = \frac{1}{d} E\{|f'(z_1)|^2 \text{ subject to } f(z_1) = 0\},$$

where by

$$E\{|f'(z_1)|^2 \text{ s.t. } f(z_1) = 0\},$$

the expected value of $|f'(z_1)|^2$ subject to $f(z_1) = 0$, we mean

$$\int_{\mathbf{C}} \psi(0, t)|t|^2 \, dt,$$

where ψ is the joint density function of $f(z_1)$ and $f'(z_1)$. One can generalize this formula to the joint density of k randomly chosen roots

$$P(z_1, \cdots, z_k) = \frac{1}{d(d-1)\cdots(d-k+1)} E\{|f'(z_1)|^2 \cdots |f'(z_k)|^2 \text{ s.t. } f(z_1) = \cdots = f(z_k) = 0\};$$

see Appendix B for the derivation. In particular,

$$P(z_1, z_2) = \frac{1}{d(d-1)} E\{|f'(z_1)|^2 |f'(z_2)|^2 \text{ s.t. } f(z_1) = (z_2) = 0\}. \tag{4.5}$$

We will estimate this expression for $\Delta z = z_2 - z_1$ with $|\Delta z| \leqq 1/d^{5/4}$ (actually $\leqq c/d$ for some constant c would give the same estimates) and $\frac{1}{2} < |z_i| < 2$.

For constants $b_0, \cdots, b_d$ we can write the random variable $\sum b_i a_i$ as $\langle \mathbf{b}, \bar{\mathbf{a}} \rangle$, where $\mathbf{b} = (b_0, \cdots, b_d) \in \mathbf{C}^{d+1}$, $\bar{\mathbf{a}} = (\bar{a}_0, \cdots, \bar{a}_d) \in \mathbf{C}^{d+1}$, $^-$ denoting complex conjugation, and $\langle\ ,\ \rangle$ denotes the usual inner product on $\mathbf{C}^{d+1}$. Analogous to sums of real normal random variables, one can easily verify that $\langle \mathbf{b}, \bar{\mathbf{a}} \rangle$ and $\langle \mathbf{b}', \bar{\mathbf{a}} \rangle$ are independent random variables if $\Re\langle \mathbf{b}, \mathbf{b}' \rangle = 0$.

For $i = 1, 2$, let $\mathbf{u}_i$ denote $(1, z_i, \cdots, z_i^d) \in \mathbf{C}^{d+1}$ and $\mathbf{v}_i$ denote

$$\mathbf{v}_i \equiv (0, 1, 2z_i, \cdots, dz_i^{d-1}) \in \mathbf{C}^{d+1}.$$

Let $\tilde{\mathbf{v}}_i$ be the projection of $\mathbf{v}_i$ onto $(\mathbf{C}\mathbf{u}_1 + \mathbf{C}\mathbf{u}_2)^{\perp}$, i.e.,

$$\tilde{\mathbf{v}}_i = \mathbf{v}_i - \sum_{j=1}^{2} \frac{\langle \mathbf{v}_i, \mathbf{u}_j \rangle}{\langle \mathbf{u}_j, \mathbf{u}_j \rangle} \mathbf{u}_j.$$

We have

$$P(z_1, z_2) = \frac{1}{\binom{d}{2}} E\{|\langle \mathbf{v}_1, \bar{\mathbf{a}} \rangle|^2 |\langle \mathbf{v}_2, \bar{\mathbf{a}} \rangle|^2 \text{ s.t. } \langle \mathbf{u}_1, \bar{\mathbf{a}} \rangle = \langle \mathbf{u}_2, \bar{\mathbf{a}} \rangle = 0\}$$

$$= \frac{1}{\binom{d}{2}} E\{|\langle \tilde{\mathbf{v}}_1, \bar{\mathbf{a}} \rangle|^2 |\langle \tilde{\mathbf{v}}_2, \bar{\mathbf{a}} \rangle|^2 \text{ s.t. } \langle \mathbf{u}_1, \bar{\mathbf{a}} \rangle = \langle \mathbf{u}_2, \bar{\mathbf{a}} \rangle = 0\}$$

[3] Equivalently, the chance of finding at least one root in $B_\varepsilon(z_1)$ is $dP(z_1) \cdot \pi\varepsilon^2 +$ (lower order terms).

(since $\langle \mathbf{v}_j, \bar{\mathbf{a}} \rangle = \langle \tilde{\mathbf{v}}_j, \bar{\mathbf{a}} \rangle$ if $\langle \mathbf{u}_i, \bar{\mathbf{a}} \rangle = 0$)

$$\frac{1}{\binom{d}{2}} E\{|\langle \tilde{\mathbf{v}}_1, \bar{\mathbf{a}} \rangle|^2 |\langle \tilde{\mathbf{v}}_2, \bar{\mathbf{a}} \rangle|^2\} \psi(0,0) \tag{4.6}$$

by independence, where ψ is the joint density of $\langle \mathbf{u}_1, \bar{\mathbf{a}} \rangle$, $\langle \mathbf{u}_2, \bar{\mathbf{a}} \rangle$. Similar to the case of real normal random variables,

$$\psi(0,0) = \left(\frac{1}{2\pi}\right)^2 \left| \det \begin{pmatrix} \langle \mathbf{u}_1, \mathbf{u}_1 \rangle & \langle \mathbf{u}_1, \mathbf{u}_2 \rangle \\ \langle \mathbf{u}_2, \mathbf{u}_1 \rangle & \langle \mathbf{u}_2, \mathbf{u}_2 \rangle \end{pmatrix} \right|^{-1}$$

(see Appendix A for details). Letting $\Delta\mathbf{u} = \mathbf{u}_2 - \mathbf{u}_1$, we have

$$\begin{aligned} \Delta\mathbf{u} &= (z_2 - z_1)(0, 1, z_1 + z_2, \cdots, z_1^{d-1} + z_1^{d-2} z_2 + \cdots + z_2^{d-1}) \\ &= \Delta z (0, 1, 2z_1, \cdots, d z_1^{d-1})(1 + O(d^{-1/4})) \\ &= \Delta z \mathbf{v}_1 (1 + O(d^{-1/4})) \end{aligned}$$

and so

$$\begin{aligned} \left| \det \begin{pmatrix} \langle \mathbf{u}_1, \mathbf{u}_1 \rangle & \langle \mathbf{u}_1, \mathbf{u}_2 \rangle \\ \langle \mathbf{u}_2, \mathbf{u}_1 \rangle & \langle \mathbf{u}_2, \mathbf{u}_2 \rangle \end{pmatrix} \right| &= \left| \det \begin{pmatrix} \langle \mathbf{u}_1, \mathbf{u}_1 \rangle & \langle \mathbf{u}_1, \Delta\mathbf{u} \rangle \\ \langle \Delta\mathbf{u}, \mathbf{u}_1 \rangle & \langle \Delta\mathbf{u}, \Delta\mathbf{u} \rangle \end{pmatrix} \right| \\ &= |\Delta z|^2 (|\mathbf{u}_1|^2 |\mathbf{v}_1|^2 (1 + O(d^{-1/4})) - |\langle \mathbf{u}_1, \mathbf{v}_1 \rangle|^2 (1 + O(d^{-1/4}))). \end{aligned}$$

We have

$$\begin{aligned} |\mathbf{u}_1|^2 &= 1 + |z_1|^2 + \cdots + |z_1|^{2d}, \\ |\mathbf{v}_1|^2 &= 1 + 4|z_1|^2 + \cdots + d^2 |z_1|^{2d-2}, \\ \langle \mathbf{u}_1, \mathbf{v}_1 \rangle &= z_1 (1 + 2|z_1|^2 + 3|z_1|^4 + \cdots + d|z_1|^{2d-2}), \\ |\langle \mathbf{u}_1, \mathbf{v}_1 \rangle|^2 &= |z_1|^2 (1 + 2|z_1|^2 + 3|z_1|^4 + \cdots + d|z_1|^{2d-2})^2. \end{aligned}$$

PROPOSITION 4.4. *$|\mathbf{u}_1|^2 |\mathbf{v}_1|^2$, $|\langle \mathbf{u}_1, \mathbf{v}_1 \rangle|^2$, and $|\mathbf{u}_1|^2 |\mathbf{v}_1|^2 - |\langle \mathbf{u}_1, \mathbf{v}_1 \rangle|^2$ are each $\Theta(d^4)$ for $1 - (1/d) \leqq |z| \leqq 1$ and $\Theta((1 - |z|^2)^{-4})$ for $|z| \leqq 1 - (1/d)$. (For f to be $\Theta(g)$ means $c_1 g < f < c_2 g$ for some constants c_1 and c_2.)*

Proof. Let $y = |z|^2$. We have

$$\begin{aligned} |\mathbf{u}_1|^2 |\mathbf{v}_1|^2 &= (1 + y + \cdots + y^d)(1 + 4y + \cdots + d^2 y^{d-1}), \\ |\langle \mathbf{u}_1, \mathbf{v}_1 \rangle|^2 &= y(1 + 2y + \cdots + d y^{d-1})^2, \end{aligned}$$

and upon subtraction

$$\begin{aligned} |\mathbf{u}_1|^2 |\mathbf{v}_1|^2 - |\langle \mathbf{u}_1, \mathbf{v}_1 \rangle|^2 = 1 &+ \binom{4}{3} y + \binom{5}{3} y^2 + \cdots + \binom{d+2}{3} y^{d-1} \\ &+ c_d y^d + c_{d+1} y^{d+1} + \cdots + c_{2d-2} y^{2d-2}, \end{aligned}$$

where $c_d, \cdots, c_{2d-2}$ are positive integers. If $1 - (1/d) < |z| < 1$, then we have $e^{-2} \leqq |z|^j \leqq 1$ for any $j = 1, \cdots, 2d-1$ and the aforementioned estimates easily follow. If $|z| < 1 - (1/d)$, then the proposition follows using

$$\sum_{i=0}^{d} i^n r^i \text{ and } \sum_{i=0}^{\infty} i^n r^i \text{ are both } = \Theta\left(\frac{1}{1-r}\right)^n$$

for any n and any $r<1-c/d$ for any fixed c (this latter condition ensures that the former sum has enough terms to approximate its limiting infinite sum).

COROLLARY 4.5.

$$\psi(0,0)\leqq\frac{c}{|\Delta z|^2}\begin{cases}d^4 & \text{if } 1-\frac{1}{d}\leqq|z_1|\leqq 1\\ \dfrac{d^2}{(1-|z_1|^2)^4} & \text{if } \frac{1}{2}\leqq|z_1|\leqq 1-\frac{1}{d}.\end{cases}$$

To estimate

$$E\{|\langle\tilde{\mathbf{v}}_1,\bar{\mathbf{a}}\rangle|^2|\langle\tilde{\mathbf{v}}_2,\bar{\mathbf{a}}\rangle|^2\}$$

note that

$$\mathbf{u}_2=\mathbf{u}_1+(\Delta z)\mathbf{v}_1+\frac{(\Delta z)^2}{2}\mathbf{w}_1(1+O(d^{-1/4}))$$

where

$$\mathbf{w}_1=(0,0,2,6z_1,\cdots,d(d-1)z_1^{d-2}).$$

Hence

$$\tilde{\mathbf{v}}_1=\left(\mathbf{v}_1-\frac{\mathbf{u}_2-\mathbf{u}_1}{\Delta\mathbf{z}}\right)^{\sim}=\frac{\Delta z}{2}\tilde{\mathbf{w}}_1(1+O(d^{-1/4}))$$

(where $\tilde{}$ denotes the projection onto $(\mathbf{Cu}_1+\mathbf{Cu}_2)^{\perp}$). Similarly we have

$$\tilde{\mathbf{v}}_2=\frac{\Delta z}{2}\tilde{\mathbf{w}}_1(1+O(d^{-1/4})).$$

So we estimate

$$E\{|\langle\tilde{\mathbf{v}}_1,\bar{\mathbf{a}}\rangle|^2|\langle\tilde{\mathbf{v}}_2,\bar{\mathbf{a}}\rangle|^2\}\leqq c(\Delta z)^4E\{|\langle\tilde{\mathbf{w}}_1,\bar{\mathbf{a}}\rangle|^2|\langle\tilde{\mathbf{w}}_2,\bar{\mathbf{a}}\rangle|^2\}$$

(where c is an absolute constant replacing $(1+O(d^{-1/4}))$

$$\leqq c(\Delta z)^4(E\{|\langle\tilde{\mathbf{w}}_1,\bar{\mathbf{a}}\rangle|^4\}+E\{|\langle\tilde{\mathbf{w}}_1,\bar{\mathbf{a}}\rangle|^4\})$$

by Schwartz's inequality. To simplify estimating these fourth moments we use Proposition 4.6.

PROPOSITION 4.6. *Let α, β, and γ be independent complex valued random variables with $E\{\alpha^i\bar{\alpha}^j\}=0$ for $i\neq j$ and similarly for β and γ. Then*

$$E\{|\alpha|^4\}\leqq E\{|\alpha+\beta+\gamma|^4\}.$$

Proof.

$$E\{|\alpha+\beta+\gamma|^4\}=E\{(\alpha+\beta+\gamma)^2(\bar{\alpha}+\bar{\beta}+\bar{\gamma})^2\},$$

which, when expanded as the sum of expectations of products, has terms that are of the form $E\{\delta\bar{\delta}\}>0$ or that drop out. One of these terms is $E\{\alpha^2\bar{\alpha}^2\}$.

Since $\langle\mathbf{w}_1,\bar{\mathbf{a}}\rangle$ is the sum of the three independent, radially symmetric random variables $\langle\tilde{\mathbf{w}}_1,\bar{\mathbf{a}}\rangle$, $\alpha_1\langle\mathbf{u}_1,\bar{\mathbf{a}}\rangle$, and $\alpha_2\langle\mathbf{u}_2,\bar{\mathbf{a}}\rangle$ for appropriate α_1, α_2, we have

$$E\{|\langle\tilde{\mathbf{w}}_1,\bar{\mathbf{a}}\rangle|^4\}\leqq E\{|\langle\mathbf{w}_1,\bar{\mathbf{a}}\rangle|^4\}.$$

Now

$$E\{|\langle\mathbf{w}_1,\bar{\mathbf{a}}\rangle|^4\}=E\left\{\left|\sum_{i=0}^{d}i(i-1)z^{i-2}a_i\right|^4\right\}$$

$$=\sum_{i,j,k,l}i(i-1)j(j-1)k(k-1)l(l-1)z^{i+j+k+l-8}E\{a_ia_j\bar{a}_k\bar{a}_l\}.$$

The only terms not vanishing in the latter sum are those for which either $i=k$, $j=l$, or $i=l$, $j=k$. By the symmetry of these conditions, and since the $E\{a_i a_j \bar{a}_k \bar{a}_l\}$ are bounded, we can estimate the above sum by

$$\leqq c \sum_{i,j} i^4 j^4 |z|^{2(i+j)-8} \leqq c' \sum_{m=0}^{2d} m^9 |z|^{2m-8}, \tag{4.7}$$

where we have set $m=i+j$. If $|z|<1-(1/d)$, we can estimate (4.7) by using

$$\sum_{i=0}^{\infty} i^9 r^i \leqq c\left(\frac{1}{1-r}\right)^{10}$$

to get

$$E\{|\langle \tilde{\mathbf{w}}_1, \bar{\mathbf{a}}\rangle|^4\} \leqq c\left(\frac{1}{1-|z|^2}\right)^{10},$$

which gives $O(d^{10})$ or better for $|z|$ in this range. For $1-(1/d)\leqq |z|\leqq 1$ we simply use $|z|^m \leqq 1$ in (4.7) to get

$$E\{|\langle \tilde{\mathbf{w}}_1, \bar{\mathbf{a}}\rangle|^4\} \leqq c \sum_{m=0}^{2d} m^9 \leqq c' d^{10}.$$

Summing up, we have the following lemma.

LEMMA 4.7.

$$E\{|\langle \tilde{v}_1, \bar{a}\rangle|^2 |\langle \tilde{v}_2, \bar{a}\rangle|^2\} \leqq c|\Delta z|^4 \begin{cases} d^{10} & \text{if } 1-\dfrac{1}{d} \leqq |z_1| \leqq 1 \\ (1-|z_1|^2)^{-10} & \text{if } \dfrac{1}{2} \leqq |z_1| \leqq 1-\dfrac{1}{d}. \end{cases}$$

Combining Lemma 4.7, Corollary 4.5, and (4.6) yields Theorem 4.8.

THEOREM 4.8.

$$P(z_1, z_2) \leqq \frac{c}{d(d-1)} (\Delta z)^2 \begin{cases} d^6 & \text{if } 1-\dfrac{1}{d} \leqq |z_1| \leqq 1 \\ (1-|z_1|^2)^{-6} & \text{if } \dfrac{1}{2} \leqq |z_1| \leqq 1-\dfrac{1}{d}. \end{cases}$$

COROLLARY 4.9. *For $1\leqq |z_1|\leqq 2$, the same estimates, as in Theorem* 4.8, *hold (with slightly different c and θ).*

Proof. Let $y_1 = 1/z_1$, $y_2 = 1/z_2$. Let $\tilde{P}(\,,\,)$ be the density of two random roots of

$$a_0 y^d + a_1 y^{d-1} + \cdots + a_d = 0. \tag{4.8}$$

On the one hand, clearly $\tilde{P} = P$. On the other hand, y satisfies (4.8) if and only if $x = 1/y$ satisfies

$$a_d x^d + \cdots + a_0 = 0.$$

Thus

$$\begin{aligned} P(s,t) &= \tilde{P}\left(\frac{1}{s}, \frac{1}{t}\right) \left|\frac{\partial(1/s, 1/t)}{\partial(s,t)}\right|^2 \\ &= \tilde{P}\left(\frac{1}{s}, \frac{1}{t}\right) \frac{1}{|st|^4} \\ &= P\left(\frac{1}{s}, \frac{1}{t}\right) \frac{1}{|st|^4}. \end{aligned}$$

Thus, since $1 \leqq |z_1|, |z_2| \leqq 2$, we have

$$P(z_1, z_2) \leqq P(y_1, y_2) \frac{1}{4^4}.$$

Since $|y_1 - y_2| \leqq c/d$ can be ensured by requiring $|z_1 - z_2| \leqq c'd$, we can apply Theorem 4.8 in this case to obtain the desired estimate (note that $(1-|z_1|)^2$ and $(1-|y_1|)^2$ differ from each other by some multiple in a bounded, positive range).

LEMMA 4.10.

$$\Pr\{|z_1 - z_2| \leqq \delta \text{ and } \tfrac{1}{2} \leqq |z_1| \leqq 2\} \leqq c\delta^4 d^3.$$

Proof. We have

$$\int_{t \in B_2(0) - B_{1/2}(0)} \int_{s \in B_\delta(t)} P(s, t)\, ds\, dt$$

$$\leqq \int_{t \in B_2(0) - B_{1/2}(0)} \frac{c}{d^2} \delta^4 \begin{cases} d^6 & \text{if } 1 - \frac{1}{d} \leqq |t| \leqq 1 \\ (1-|t|2)^{-6} & \text{if } \frac{1}{2} \leqq |t| \leqq 1 - \frac{1}{d} \end{cases} dt. \tag{4.9}$$

The integral in (4.9), over the range $1-(1/d) \leqq |t| \leqq 1+(1/d)$ is

$$\leqq \frac{c}{d^2} \delta^4 d^6 |B_{1+(1/d)}(0) - B_{1-(1/d)}(0)|$$

$$= \frac{c}{d^2} \delta^4 d^6 \frac{4\pi}{d} = c'\delta^4 d^3.$$

Over the range $\frac{1}{2} \leqq |t| \leqq 1-(1/d)$, setting $r = |t|$, the integral of (4.9) becomes

$$\int_t \frac{c}{d^2} \delta^4 \left(\frac{1}{1-|t|^2}\right)^6 dt = \int_{r=1/2}^{1-(1/d)} \frac{c}{d^2} \delta^4 \left(\frac{1}{1-r^2}\right)^6 2\pi r\, dr$$

$$= \frac{c'}{d^2} \delta^4 (1-r^2)^{-5} \Big|_{1/2}^{1-(1/d)}$$

$$= \frac{c'}{d^2} \delta^4 \left[\left(\frac{2}{d} - \frac{1}{d^2}\right)^{-5} - \left(\frac{3}{4}\right)^{-5}\right] \leqq c''\delta^4 d^3.$$

COROLLARY 4.11. *The probability that there is a root $|z_i|$ with $\frac{1}{2} \leqq |z_i| \leqq 2$ for which there is some other root, $|z_j|$ with $|z_i - z_j| \leqq \delta$, is no more than $c\delta^4 d^5$ for some constant c.*

Proof. For $\delta > d^{-5/4}$ the statement holds with $c = 1$. For $\delta \leqq d^{-5/4}$ we apply Lemma 4.10 to each of the $d(d-1)$ pairs of roots, z_i, z_j, $i \neq j$; the total probability is no more than the sum of the $d(d-1)$ probabilities $c'\delta^4 d^3$.

Finally we arrive at our main theorem.

THEOREM 4.12. *For a_i's distributed as independent standard normals,*

$$\Pr\{Q_d(\varepsilon)\} \leqq c\varepsilon^2 d^7 + 2^{-c'd}$$

for some constants c, $c' > 0$.

Proof. By Corollary 4.11 and Corollary 4.3 we have that with probability $\geqq 1 - c\delta^4 d^3 - 2^{-c'd}$ we have at least $d/2$ of the roots z lying in the range $\frac{1}{2} \leqq |z| \leqq 2$ and each such root is separated from the others by a distance δ. By Lemma 2.1 this guarantees for each of these $d/2$ roots an approximate zero region of area $\geqq c\delta^2/(d-1)^2$, for a total of $\geqq c\delta^2/d$. Setting $\varepsilon = c\delta^2/d$ we get $\delta^4 d^5 = c\varepsilon^2 d^7$ and the theorem follows.

5. Uniform and some other distributions. In this section we obtain estimates like those of the previous section for coefficients distributed independently according to some other distribution. We will assume the distribution is the uniform distribution in $B_1(0)$ for a_i with $i<d$ and $a_d=1$. In fact, one can do the same estimtes verbatim with $a_d=1$ and for $i<d$ taking a_i according to any distribution supported in $B_1(0)$, possibly different for different i's, which satisfy (5.3) for $l=14$ with uniform bounds on the $m_i's$.

In Smale's works, [Sma81] and [Sma86b], the polynomials $a_dz^d+\cdots+a_0$ are considered with $a_d=1$ and a_i distributed uniformly in $B_1(0)$ for $i\neq d$. This has the advantage of guaranteeing that the roots lie in the ball of radius 2 (if $|z|>2$, then clearly $|z^d|>\sum_{i<d}|a_iz^i|$ if $|a_i|\leqq 1$ and so such a z cannot be a root of the polynomial). In contrast to the distribution of the previous section, (almost all) such polynomials have regions of approximate zeros in $B_3(0)$, and we will obtain estimates of the form

$$\Pr\{Q(\varepsilon)\}\leqq cd^7\varepsilon^2. \tag{5.1}$$

We begin by considering the probability measure on polynomials in which $a_d=1$ and the remaining coefficients distributed uniformly in the unit ball, $B_1(0)$; i.e., with density

$$\psi(z)=\begin{cases}1/\pi & \text{for } z\in B_1(0)\\ 0 & \text{otherwise.}\end{cases}$$

For the density ψ, we have its characteristic function, $\hat{\psi}$, which satisfies

$$|\hat{\psi}(\xi)|\leqq\frac{k}{|\xi|}\ \forall\xi\in\mathbf{C} \tag{5.2}$$

for some k, and

$$\hat{\psi}(\xi)=1-m_1|\xi|^2-m_2|\xi|^4-\cdots-m_l|\xi|^{2l}+O(|\xi|^{2l+2}) \tag{5.3}$$

for any l (see Appendix A.)

It will be easier to have all the a_i's radially symmetric, so we will take a_d to be distributed as

$$e^{2\pi i\theta}$$

with θ uniform random variable in $[0,1]$. We denote its characteristic function by

$$\hat{\psi}_1(\xi)=E\{e^{i\Re(\langle\xi,a_d\rangle)}\}.$$

For $\hat{\psi}_1$ we also have an expansion

$$\hat{\psi}_1(\xi)=1-M_1|\xi|^2-M_2|\xi|^4-\cdots-M_l|\xi|^{2l}+O(|\xi|^{2l+2}).$$

We begin by estimating $P(z_1,z_2)$ for $|z_1-z_2|$ small. As in the previous section, by $|z_1-z_2|$ *small* it suffices to take $|z_1-z_2|\leqq c|z_1|/d$ for some constant c, but we will only be applying the estimate when $|z_1-z_2|\leqq c|z_1|d^{-5/4}$; we will assume the latter for notational convenience.

To estimate $P(z_1,z_2)$, from (4.5) we see that it suffices to estimate

$$E\{|f'(z_1)|^4 \text{ s.t. } f(z_1)=f(z_2)=0\}. \tag{5.4}$$

We can estimate it as

$$\leqq\frac{c|\Delta z|^2}{d(d-1)}\int_{\mathbf{C}}|t|^4\Upsilon(0,0,t)\,dt \tag{5.5}$$

where Υ is the joint density of

$$\sum a_i u_i, \sum a_i v_i, \sum a_i w_i \tag{5.6}$$

with

$$u_i = z_1^i, \qquad v_i = i z_1^{i-1}(1+O(d^{-1/4})), \qquad w_i = i(i-1) z_1^{i-2}(1+O(d^{-1/4})).$$

Let $\tilde{a}_i$ be distributed as $\sqrt{2m_1}$ times the standard normal distribution, and Ξ the distribution of

$$\sum \tilde{a}_i u_i, \sum \tilde{a}_i v_i, \sum \tilde{a}_i w_i. \tag{5.7}$$

The main task of this section is to prove the following theorem.

THEOREM 5.1. *There exist constants c and d_0 independent of d, t, z_1, and z_2 such that the following hold. For all z_1 and z_2 with $|z_1 - z_2| \leqq |z_1|(1+O(d^{-5/4}))$, we have if $d \leqq d_0$ or $|z_1| \leqq \frac{1}{2}$ then*

$$\int_{\mathbf{C}} |t|^4 \Upsilon(0, 0, t)\, dt \leqq c, \tag{5.8}$$

if $d > d_0$ and $1-(1/d) \leqq |z_1| \leqq 1+(1/d)$ then

$$\Upsilon(0, 0, t) \leqq c\Xi(0, 0, t) + c\Xi(0, 0, t/2) + cd^{-15}, \tag{5.9}$$

and if $d > d_0$ and $\frac{1}{2} \leqq |z_1| \leqq 1-(1/d)$ then

$$\Upsilon(0, 0, t) \leqq c\Xi(0, 0, t) + c\Xi(0, 0, t/2) + c(1-|z_1|^2)^{-12}. \tag{5.10}$$

This will give estimates on (5.5), and thus on (5.4), similar to those in § 4.

For (5.8), notice that for any d_0 there is a c such that for any $d \leqq d_0$ or if $|z_1| \leqq \frac{1}{2}$, we have

$$\int |t|^4 \Upsilon(0, 0, t)\, dt = E\{|\textstyle\sum a_i w_i|^4 \text{ s.t. } \sum a_i u_i = \sum a_i v_i = 0\}$$

$$\leqq \max |\textstyle\sum a_i w_i|^4 \Upsilon(0, 0) \leqq c$$

where $\Upsilon(0, 0)$ is the joint density of $\sum a_i u_i$, $\sum a_i v_i$ at $(0, 0)$, since

$$\sum |a_i||v_i| \leqq \sum |v_i|,$$

which is bounded uniformly for such $|z_1|$, and

$$\sum a_i u_i = a_0 + z_1 a_1 + \cdots$$

$$\sum a_i v_i = a_1 + \cdots$$

so that

$$\begin{aligned} &\Pr\{\textstyle\sum a_i u_i \in B_\varepsilon(0), \sum a_i v_i \in B_\varepsilon(0)\} \\ &= \Pr\{a_0 \in B_\varepsilon(l_0),\, a_1 \in B_\varepsilon(l_1)\} \\ &\leqq \varepsilon^4, \end{aligned}$$

where l_0, l_1 are linear combinations of $a_2, \cdots, a_d$, and so

$$\Upsilon(0, 0) \leqq \left(\frac{1}{\pi}\right)^2.$$

Next we estimate $\Upsilon(0, 0, t)$ for $1 \leqq |z_1| \leqq 1+(1/d)$.

It will be convenient to rescale u, v, and w via

$$U_i \equiv \frac{u_i}{z_1^d}, \qquad V_i \equiv \frac{v_i}{dz_1^{d-1}}, \qquad W_i \equiv \frac{w_i}{d(d-1)z_1^{d-2}},$$

and set

$$Y_i \equiv (U_i, V_i, W_i) \approx z_1^{i-d}\left(1, \frac{i}{d}, \frac{i^2}{d^2}\right).$$

Note that $|Y_i| \leqq c$ for some constant c independent of i and d. We will obtain estimates as in (5.9), with Ψ being the density of

$$\sum a_i U_i, \sum a_i V_i, \sum a_i W_i \tag{5.11}$$

and Φ the density of (5.11) with a_i replaced by $\tilde{a}_i$.

Consider, for each j, $(a_jU_j, a_jV_j, a_jW_j) \in \mathbf{C}^3 \simeq \mathbf{R}^6$. Since the characteristic function of a_j for $j > d$ is

$$\hat{\psi}(\xi) = E\{e^{i\Re(\xi\bar{a}_j)}\},$$

the characteristic function of (a_jU_j, a_jV_j, a_jW_j) is

$$\begin{aligned}\tilde{\chi}(\eta, \sigma, \tau) &= E\{e^{i\Re(\eta\overline{a_juUj} + \sigma\overline{a_jV_j} + \tau\overline{a_jW_j})}\} \\ &= E\{e^{i\Re[(\eta\bar{U}_j + \sigma\bar{V}_j + \tau\bar{W}_j)\bar{a}_j]}\} \\ &= \hat{\psi}(|\eta\bar{U}_j + \sigma\bar{V}_j + \tau\bar{W}_j|).\end{aligned}$$

It follows that the characteristic function of the joint density of (5.6) is

$$\hat{\Psi} = \hat{\psi}_1(|\eta\bar{U}_j + \sigma\bar{V}_j + \tau\bar{W}_j|) \prod_{j=0}^{d-1} \hat{\psi}(|\eta\bar{U}_j + \sigma\bar{V}_j + \tau\bar{W}_j|),$$

and its density is

$$\Psi(q, s, t) = \left(\frac{1}{2\pi}\right)^6 \int_{\mathbf{C}^3} e^{-i\Re(q\bar{\eta} + s\bar{\sigma} + t\bar{\tau})}\hat{\Psi}(\eta, \sigma, \tau)\, d\eta\, d\sigma\, d\tau.$$

Let $\xi = (\eta, \sigma, \tau)$. Then

$$\hat{\Psi}(\xi) = \hat{\psi}_1(|\langle \xi, Y_d\rangle|) \prod_{i=0}^{d-1} \hat{\psi}(|\langle \xi, Y_i\rangle|).$$

From (5.3) it follows that

$$\hat{\psi}(\xi) = e^{-m_1|\xi|^2 + m_2'|\xi|^4 + \cdots + m_7'|\xi|^{14}} + O(|\xi|^{16})$$

and

$$\hat{\psi}_1(\xi) = e^{-M_1|\xi|^2 + M_2'|\xi|^4 + \cdots + M_7'|\xi|^{14}} + O(|\xi|^{16})$$

for $|\xi|$ small for some constants $m_2', \cdots, m_7'$ and $M_2', \cdots, M_7'$. Let δ be a small positive number with $22\delta \leqq 1$. Let

$$B \equiv B_{d^{\delta - 1/2}}(0).$$

Let $\hat{\omega}$ be defined by

$$\hat{\omega}(\xi) = \begin{cases} e^{-m|\xi|^2 + m_2'|\xi|^4 + \cdots + m_7'|\xi|^{14}} & \text{for } z \in B \\ 0 & \text{otherwise,} \end{cases}$$

and let $\hat{\omega}_1$ be defined by replacing the m's by M's. Let Ω be given by

$$\hat{\Omega}(\xi) = \hat{\omega}_1(|\langle \xi, Y_d\rangle|) \prod_{i=0}^{d-1} \hat{\omega}(|\langle \xi, Y_i\rangle|).$$

Henceforth we will replace ϕ_1 and ω_1 by ψ and ω when we write our equations. It makes no difference in the analysis.

We reduce the study of Ψ to that of Ω.

LEMMA 5.2. *For some constant c we have for any z_1 with $1 \leqq |z_1| \leqq 1 + (1/d)$,*

$$|\Psi(0, 0, t) - \Omega(0, 0, t)| \leqq cd^{-9} \quad \forall t \in \mathbf{C}.$$

Proof. By the Fourier inversion formula,

$$\begin{aligned} |\Psi(0, 0, t) - \Omega(0, 0, t)| &\leqq \left(\frac{1}{2\pi}\right)^6 \int_{\mathbf{C}^3} |[\hat{\Psi}(\xi) - \hat{\Omega}(\xi)]\, e^{-i\Re\langle x, \xi\rangle}|\, d\xi \\ &\leqq \left(\frac{1}{2\pi}\right)^6 \int_{\mathbf{C}^3} |\hat{\Psi}(\xi) - \hat{\Omega}(\xi)|\, d\xi \qquad (5.12) \\ &\leqq \left(\frac{1}{2\pi}\right)^6 \left[\int_{\mathbf{C}^3 - B} |\hat{\Psi}(\xi)|\, d\xi + \int_B |\hat{\Psi}(\xi) - \hat{\Omega}(\xi)|\, d\xi\right]. \end{aligned}$$

To estimate the second integral of (5.12), we have

$$\int_B |\hat{\Psi} - \hat{\Omega}|\, d\xi = \int_B \left| \prod_{i=0}^{d} \hat{\psi}(\langle \xi, Y_i\rangle) - \prod_{i=0}^{d} \hat{\omega}(\langle \xi, Y_i\rangle) \right| d\xi,$$

which, by Lemma A.1, is

$$\leqq \int_B \sum |\hat{\psi}(\langle \xi, Y_i\rangle) - \hat{\omega}(\langle \xi, Y_i\rangle)|\, d\xi \leqq c \int_B \sum |\langle \xi, Y_i\rangle|^{16}\, d\xi,$$

which, by Cauchy-Schwarz and since the $|Y_i|$ are bounded independent of i and d,

$$\begin{aligned} &\leqq c \int_B \sum |\xi|^{16}\, d\xi \leqq c|B|\, d(d^{\delta - 1/2})^{16} = c(d^{\delta - 1/2})^{22} d \qquad (5.13) \\ &= cd^{22\delta - 10} \leqq cd^{-9} \end{aligned}$$

since $22\delta \leqq 1$, where we have used the fact that $|B_r(0)| = cr^6$ for balls in $\mathbf{C}^3$.

To estimate the first integral of (5.12), let

$$\begin{aligned} D_0 &\equiv \{i \in \mathbf{Z}: \tfrac{5}{6}d - 2 \leqq i < d\}, \\ D_1 &\equiv \{i \in \mathbf{Z}: \tfrac{3}{6}d - 1 \leqq i \leqq \tfrac{4}{6}d\}, \\ D_2 &\equiv \{i \in \mathbf{Z}: \tfrac{1}{6}d - 1 \leqq i \leqq \tfrac{2}{6}d\}. \end{aligned}$$

Note that for d sufficiently large the D_j's are disjoint and each D_j contains $\geqq d/6$ integers. Let d_0 be an integer such that this is the case for $d > d_0$. This will be our d_0 in Theorem 5.1. We will use the following sublemma.

SUBLEMMA 5.3. *Let $i \in D_0$, $j \in D_1$, $k \in D_2$. Then for each $\xi \in \mathbf{C}^3$, either*

$$|\langle \xi, Y_i\rangle|, \qquad |\langle \xi, Y_j\rangle|, \text{ or } |\langle \xi, Y_k\rangle| \text{ is } \geqq c_2|\xi|$$

for some absolute constant c_2 (independent of ξ and z_1 with $1 \leqq |z_1| \leqq 1 + (1/d)$).

Proof. This is an easy calculation. For d small one can use compactness in $z_1 \in B_{1+(1/d)}(0) - B_1(0)$. For d large we have

$$Y_i \approx z_1^{i-d}(1, \theta, \theta^2)$$

with $(1/6)-(1/d)\leqq\theta\leqq(2/6)$; note that $z_1^{i-d}\in B_1(0)-B_{1/e}(0)$, since $|z_1|^{-d}\geqq(1+(1/d))^{-d}\geqq 1/e$. Similar estimates hold for Y_j and Y_k, in which θ takes on a different range of values, leading to the desired estimate.

Let $A=B_{2kc_2}(0)$ where c_2 is the constant in Lemma 5.3 and k is the constant in Lemma 5.2. We write

$$\int_{\mathbf{C}^3-B}|\hat{\Psi}(\xi)|\,d\xi=\int_{\mathbf{C}^3-A}|\hat{\Psi}(\xi)|\,d\xi+\int_{A-B}|\hat{\Psi}(\xi)|\,d\xi. \tag{5.14}$$

In $\mathbf{C}^3-A$ we use

$$|\hat{\Psi}(\xi)|=\prod_{i=0}^{d}|\hat{\psi}(\langle\xi,\,Y_i\rangle)|\leqq\left(\frac{k}{c_2|\xi|}\right)^{d/6},$$

which follows from (5.2) and Lemma 5.3, to obtain

$$\int_{\mathbf{C}^3-A}|\hat{\Psi}(\xi)|\,d\xi\leqq\int_{\mathbf{C}^3-A}\left(\frac{k}{c_2|\xi|}\right)^{d/6}d\xi\leqq c\left(\frac{1}{2}\right)^{d/6}. \tag{5.15}$$

In $A-B$, we take a constant k'' with the property that

$$|\hat{\psi}(\xi)|\leqq e^{-|\xi|^2k''}\quad\forall\xi\in A,$$

and estimate

$$|\hat{\Psi}(\xi)|\leqq\prod_{i=0}^{d}e^{-k''|\langle\xi,Y_i\rangle|^2}\leqq e^{-k''(d/6)|\xi|^2c_2^2}=e^{-cd|\xi|^2}$$

so that

$$\int_{A-B}|\hat{\Psi}(\xi)|\,d\xi\leqq\int_{A-B}e^{-cd|\xi|^2}\,d\xi\leqq c'\,e^{-cd^{2\delta}}. \tag{5.16}$$

Combining (5.12), (5.13), (5.14), (5.15), and (5.16) yields Lemma 5.2.

To deal with Ω, note that for $\xi\in B$,

$$\hat{\Omega}(\xi)=e^{-Q_2(\xi)-\cdots-Q_{14}(\xi)},$$

where the Q_i are homogeneous polynomials of degree i in $\xi=(\eta,\sigma,\tau)$ given by

$$Q_{2k}(\xi)=\sum_{i=0}^{d}m_k'|\langle\xi,\,Y_i\rangle|^{2k},$$

where $m_1'=m_1$. Note that

$$Q_2(\xi)\geqq m_1c_2^2\frac{d}{6}|\xi|^2\geqq cd|\xi|^2$$

and that

$$|Q_{2k}(\xi)|\leqq\sum_{i=0}^{d}m_k'|\xi|^{2k}|\,Y_i|^{2k}\leqq cd|\xi|^{2k}.$$

Expanding by power series we get

$$\begin{aligned}\hat{\Omega}(\xi)&=e^{-Q_2(\xi)-\cdots-Q_{14}(\xi)}\\&=e^{-Q_2(\xi)}(1+R_4(\xi)+\cdots+R_{14}(\xi))+O(|\xi|^{16}),\end{aligned}$$

where $R_k(\xi)$ are homogeneous polynomials of degree $2k$ in ξ. Since

$$|Q_{2k}(\xi)| \leqq cd^{k/2}|\xi|^{2k}$$

for $2k \geqq 4$, we have

$$|R_{2k}(\xi)| \leqq c'd^{k/2}|\xi|^{2k} \tag{5.17}$$

for some c', and hence

$$|R_{2k}(\xi)| \leqq cd^3|\xi|^{2k} \tag{5.18}$$

for $k = 2, \cdots, 7$ (in (5.17), we can replace fractional powers of d by the nearest lower integral power of d).

Let Θ be given by

$$\hat{\Theta}(\xi) = e^{-Q_2(\xi)}(1 + R_4(\xi) + \cdots + R_{14}(\xi)).$$

We finish the proof of Lemma 5.2 with the following lemma.

LEMMA 5.4.

$$|\Omega(0, 0, t) - \Theta(0, 0, t)| \leqq cd^{-9}.$$

LEMMA 5.5.

$$|\Theta(0, 0, t)| \leqq c\Phi(0, 0, t).$$

The proof of Lemma 5.4 is the same as the proof of Lemma 5.2. Note

$$\int_{\mathbf{C}^3} |\hat{\Omega} - \hat{\Theta}|\, d\xi = \int_{\mathbf{C}^3 - B} |\hat{\Theta}|\, d\xi + \int_B |\hat{\Omega} - \hat{\Theta}|\, d\xi.$$

Similar to the proof of Lemma 5.2, we can estimate

$$\int_B |\hat{\Omega} - \hat{\Theta}|\, d\xi \leqq cd^{-9}$$

and estimate

$$\begin{aligned}\int_{\mathbf{C}^3 - B} |\hat{\Theta}|\, d\xi &\leqq \int_{\mathbf{C}^3 - A} + \int_{A - B} \\ &\leqq cd^3\left[\left(\frac{1}{2}\right)^{d/6} + e^{-c'd^{2\delta}}\right],\end{aligned}$$

the d^3 coming from (5.18).

The proof of Lemma 5.5 is a straightforward calculation. We have

$$\hat{\Phi}(\xi) = \prod_{i=0}^{d} e^{-m_1|\langle \xi, Y_i\rangle|^2} = e^{-Q_2(\xi)}.$$

It follows that

$$\Theta(x) = \left(1 + R_4\left(i\frac{\partial}{\partial x}\right) + \cdots + R_{14}\left(i\frac{\partial}{\partial x}\right)\right)\Phi(x).$$

SUBLEMMA 5.6. *Let B be an $n \times n$, complex Hermitian matrix, and let*

$$g(x) = e^{-\langle B^{-1}x, x\rangle}.$$

Then for any multi-index α we have

$$\left|\frac{\partial^\alpha g}{\partial x^\alpha}(t)\right| \leqq \|B^{-1}\|^{|\alpha|/2} c g(t/2) \quad \forall t$$

for some constant $c = c(\alpha, n)$ independent of B.

Proof. Since B is Hermitian, it suffices to prove it, assuming B is diagonal. To prove it for B diagonal it suffices to prove it for the one variable case. In the one variable case,

$$g(t) = e^{-t^2/b}$$

and

$$\frac{\partial^\alpha g}{\partial x^\alpha}(t) = \left(\frac{1}{b}\right)^{|\alpha|/2} P_\alpha\left(\frac{t}{\sqrt{b}}\right) e^{-t^2/b},$$

where P_α is a polynomial of degree α. We have

$$P_\alpha\left(\frac{t}{\sqrt{b}}\right) e^{-(3/4)t^2/b} < c(\alpha)$$

for some constant $c(\alpha)$ for each α, and thus the sublemma follows.

Taking B to be the matrix given by

$$\langle B\xi, \xi\rangle = Q_2(\xi)$$

we get that since $Q_2(\xi) \geqq cd|\xi|^2$ that $\|B^{-1}\| \leqq c/d$ and thus

$$R_{2k}\left(i\frac{\partial}{\partial x}\right)\Phi(x) \leqq cd^{k/2}\left|\left(i\frac{\partial}{\partial x}\right)^{2k}\Phi(x)\right| \leqq cd^{k/2}d^{-k}\Phi(x/2)$$

for $k \geqq 2$ and Lemma 5.5 follows.

Combining Lemmas 5.4, 5.5, and 5.2 we get

$$\Psi(0, 0, t) \leqq c\Phi(0, 0, t) + c\Phi(0, 0, t/2) + cd^{-9}$$

for all t for some c for $1 \leqq |z_1| \leqq 1 + (1/d)$. Changing from U_i, V_i, W_i to u_i, v_i, w_i sums yields the desired estimate.

The same estimates hold for $1 - (1/d) \leqq |z_1| \leqq 1$. One can see this directly, or by using the same trick as in Corollary 4.9.

Next we estimate for $\frac{1}{2} \leqq |z_1| \leqq 1 - (1/d)$. Let m be the largest integer such that

$$|z_1|^m \geqq \tfrac{1}{2}.$$

We remark that

$$m = \theta(-\log|z_1|) = \theta(1 - |z_1|) = \theta(1 - |z_1|^2)$$

(where $f = \theta(g)$ means $c_1 g \leqq f \leqq c_2 g$ for some positive constants c_1 and c_2). We claim that

$$\Upsilon(0, 0, t) \leqq c\Xi(0, 0, t) + c\Xi(0, 0, t/2) + cm^{-12},$$

which is the same as (5.10). To see this, we go through estimates similar to those for $1 \leqq |z_1| \leqq 1 + (1/d)$. We rescale

$$U_i \equiv u_i, \qquad V_i \equiv \frac{z_1 v_i}{m}, \qquad W_i \equiv \frac{z_1^2 w_i}{m^2}$$

and let

$$Y_i = (U_i, V_i, W_i).$$

Then we have

$$|Y_i| \leqq c\, e^{-c'(i/m)}$$

for positive constants c and c' independent of i and z_1. We define Ψ and Φ as before. Set

$$B \equiv B_{m^{\delta-1/2}}(0)$$

and define ω and Ω as before. As before we get

$$|\Psi(0, 0, t) - \Omega(0, 0, t)| \leqq cm^{-9}$$

using

$$\sum_{i=0}^{d} |Y_i|^{16} \leqq c \sum_{i=0}^{d} |z_1|^{16i} \leqq c \sum_{i=0}^{\infty} |z_1|^{16i} \leqq cm.$$

Next let

$$D_0 \equiv \{i \in \mathbf{Z}: m \leqq i \leqq 2m\},$$

$$D_1 \equiv \{i \in \mathbf{Z}: 3m \leqq i \leqq 4m\},$$

$$D_2 \equiv \{i \in \mathbf{Z}: 5m \leqq i \leqq 6m\}.$$

Then Sublemma 5.3 holds for these D_0, D_1, D_2. Defining Θ and Q_i and R_i as before we have

$$Q_2(\xi) \geqq cm|\xi|^2$$

and

$$|Q_{2k}(\xi)| \leqq cm|\xi|^{2k}$$

and all the estimates go through as before to yield

$$\Psi(0, 0, t) \leqq c\Psi(0, 0, t) + c\Psi(0, 0, t/2) + cm^{-9}.$$

Upon rescaling to get Υ and Ξ we get the desired result.

COROLLARY 5.7. *If* $|z_1 - z_2| \leqq |z_1| O(d^{-5/4})$, *then we have*

$$P(z_1, z_2) \leqq \frac{c}{d(d-1)} (\Delta z)^2 \begin{cases} d^6 & \text{if } 1-(1/d) \leqq |z_1| \leqq 1+(1/d) \\ (1-|z_1|^2)^{-6} & \text{for other } |z_1| \in [0, 2] \end{cases}. \tag{5.19}$$

Proof. If $1-(1/d) \leqq |z_1| \leqq 1+(1/d)$ then

$$\left|\sum a_i w_i\right| \leqq c \sum |w_i| \leqq cd^3$$

so that $\Upsilon(0, 0, t) = 0$ for $|t| > cd^3$ and so

$$\begin{aligned} \int \Upsilon(0, 0, t)|t|^4\, dt &= \int_{B_{cd^3}(0)} \Upsilon(0, 0, t)|t|^4\, dt \\ &\leqq c \int_{B_{cd^3}(0)} (\Xi(0, 0, t) + \Xi(0, 0, t/2) + d^{-12})|t|^4\, dt \\ &\leqq c \int_{\mathbf{C}} (\Xi(0, 0, t) + \Xi(0, 0, t/2))|t|^4\, dt + cd^6 \end{aligned}$$

and using the estimates on Ξ in § 4 the above is

$$\leqq cd^6.$$

For $\frac{1}{2} \leqq |z_1| \leqq 1-(1/d)$ we have

$$\sum |w_i| \leqq cm^3$$

and the same estimates as before yield the theorem. For $1+(1/d) \leqq |z_1| \leqq 2$ we use the same trick as in Corollary 4.9 and note that the same estimates hold for $1/z_1$ and the equation

$$\sum_{i=0}^{d} a_{d-i} z^i = 0.$$

(Note that $a_d = 1$ does not affect the estimates, since D_0, D_1, D_2 never contain a_0 or a_d.) For $0 \leqq |z_1| \leqq \frac{1}{2}$ we use Theorem 5.1 to get the desired result.

We can finally prove the following theorem.

THEOREM 5.8.

$$\Pr\{Q_d(\varepsilon)\} \leqq c\varepsilon^2 d^7.$$

Proof. By integrating Corollary 5.7 as in § 4 we get that for any constant k there is a constant c such that

$$\Pr\{|z_1 - z_2| \leqq \delta \text{ and } k\delta d^{5/4} \leqq |z_1| \leqq 2\} \leqq c\delta^4 d^3.$$

We need $|z_1| \geqq k\delta d^{5/4}$ to apply Corollary 5.7. By Lemma 4.1 we see that for $f(z)$ to have $\geqq d/2$ roots of absolute value $\leqq k\delta d^{5/4}$ would imply

$$(k\delta d^{5/4})^{d/2} \leqq \frac{1}{|a_0|}\sqrt{|a_0|^2 + \cdots + |a_d|^2} \leqq \frac{d+1}{|a_0|}$$

so that

$$|a_0| \leqq \frac{d+1}{(k\delta d^{5/4})^{d/2}},$$

which happens with probability

$$\leqq \frac{cd^2}{(k\delta d^{5/4})^d},$$

which is dominated by $\delta^2 d^5$ if $\delta < d^{-5/4}$ and if we take, say, $k = \frac{1}{2}$. Thus the probability that some root in $B_2(0) - B_{k\delta d^{5/4}}(0)$ is within δ of another or that there are fewer than $d/2$ roots in $B_2(0) - B_{k\delta d^{5/4}}(0)$ is $\leqq c\delta^4 d^5$. When this is not the case, then Lemma 2.1 guarantees a total approximate zero region of area $\geqq c'\delta^2 d$. Setting $\varepsilon = c''\delta^2/d$ yields the theorem.

6. A refined estimate. In this section we improve the estimates of the previous two sections by considering the joint density of three or more roots, similar to the latter part of § 3.

THEOREM 6.1. *Let $z_1, \cdots, z_k \in \mathbf{C}$ satisfy $|z_i - z_1| = |z_1| O(d^{-1-\beta})$ for some fixed β. Then*

$$P(z_1, \cdots, z_k) \leqq \frac{c}{d(d-1)\cdots(d-k+1)} \prod_{i<j} |z_j - z_i|^2 \cdot \begin{cases} d^{k(k+1)} & \text{if } 1-(1/d) \leqq |z_1| \leqq 1+(1/d) \\ (1-|z_1|)^{-k(k+1)} & \text{for other } |z_1| \in [0, 2] \end{cases}. \tag{6.1}$$

Proof. The calculations are similar to the ones done in §§ 4 and 5. We wish to estimate

$$E\{|f'(z_1)|^2 \cdots |f'(z_k)|^2 \text{ s.t. } f(z_1) = \cdots = f(z_k) = 0\}.$$

As in Theorem 5.1, it suffices to prove the theorem for $\frac{1}{2} \leqq |z_1| \leqq 1$ and d sufficiently large; for $0 \leqq |z_1| \leqq \frac{1}{2}$ and small d, the theorem will be clear from the estimates used elsewhere, and the bounded sums of the linear combinations of the a_i's involved. For $1 \leqq |z_1| \leqq 2$, we have that the $1/z_i$'s satisfy the equation with coefficients reversed and the $\frac{1}{2} \leqq |z_1| \leqq 1$ estimates can be invoked.

For convenience, let

$$m = \begin{cases} d & \text{if } 1-(1/d) \leqq |z_1| \leqq 1 \\ \lfloor -\log_2 |z_1| \rfloor & \text{if } \frac{1}{2} \leqq |z_1| < 1-(1/d) \end{cases},$$

where $\lfloor a \rfloor$ denotes the largest integer $\leqq a$.

We first deal with the case of a_i being distributed normally. It suffices to estimate

$$E\{|F(z_1)|^2 \cdots |F(z_k)|^2\}$$

and

$$\psi(0, \cdots, 0),$$

where ψ is the density of $f(z_1), \cdots, f(z_k)$ and where $F(z_i)$ is the random variable $f'(z_i)$'s projection in $(\mathbf{C}f(z_1) + \cdots + \mathbf{C}f(z_k))^{\perp}$.

For the latter, note that by setting

$$u_i = (1, z_i, z_i^2, \cdots, z_i^d)$$

we have, first of all,

$$u_1 = (1, z_1, \cdots, z_1^d).$$

Secondly, if we set $u_j' = (u_j + u_1)/(z_j - z_1)$ for $j > 1$, then we have

$$u_2' = (0, 1, 2z_1, \cdots, dz_1^{d-1})(1 + O(d^{-\beta})).$$

Next, if we set $u_j'' = (u_j' - u_2')/(z_j - z_2)$ for $j > 2$, then we have

$$u_3'' = (0, 0, 2, 6z_1, \cdots, d(d-1)z_1^{d-2})(1 + O(d^{-\beta})).$$

Continuing in this fashion, we see that

$$\det \begin{pmatrix} \langle u_1, u_1 \rangle & \cdots & \langle u_1, u_k \rangle \\ \vdots & \ddots & \vdots \\ \langle u_k, u_1 \rangle & \cdots & \langle u_k, u_k \rangle \end{pmatrix}$$

$$= \prod_{j>i} (z_j - z_i)^2 \det \begin{pmatrix} \langle u_1, u_1 \rangle & \langle u_1, u_1' \rangle & \cdots & \langle u_1, u_1^{(k-1)} \rangle \\ \langle u_1', u_1 \rangle & \langle u_1', u_1' \rangle & \cdots & \langle u_1', u_1^{(k-1)} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle u_1^{(k-1)}, u_1 \rangle & \langle u_1^{(k-1)}, u_1' \rangle & \cdots & \langle u_1^{(k-1)}, u_1^{(k-1)} \rangle \end{pmatrix}$$

where

$$u_1^{(j)} = (0, \cdots, 0, j!, \cdots, d(d-1) \cdots (d-j+1)z_1^{d-j})(1 + O(d^{-\beta})).$$

We claim that

$$\det \begin{pmatrix} \langle u_1, u_1 \rangle & \langle u_1, u_1' \rangle & \cdots & \langle u_1, u_1^{(k-1)} \rangle \\ \langle u_1', u_1 \rangle & \langle u_1', u_1' \rangle & \cdots & \langle u_1', u_1^{(k-1)} \rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle u_1^{(k-1)}, u_1 \rangle & \langle u_1^{(k-1)}, u_1' \rangle & \cdots & \langle u_1^{(k-1)}, u_1^{(k-1)} \rangle \end{pmatrix} \geqq cm^{k^2}. \tag{6.2}$$

If we expand the above determinant into a sum of the products of the entries of the matrix, each entry has size proportional to m^{k^2}. It suffices to show (6.2), with the $u_1^{(j)}$ replaced by

$$v_j = (0, \cdots, 0, j!, \cdots, d(d-1)\cdots(d-j+1)z_1^{d-j}),$$

(i.e., the old $u_1^{(j)}$'s without the error term $(1+O(d^{-\beta}))$). Next note that

$$\det \begin{pmatrix} \langle v_0, v_0\rangle & \cdots & \langle v_0, v_{k-1}\rangle \\ \vdots & \ddots & \vdots \\ \langle v_{k-1}, v_0\rangle & \cdots & \langle v_{k-1}, v_{k-1}\rangle \end{pmatrix} = \det \begin{pmatrix} \langle v_0, v_0\rangle & \langle v_0, \tilde{v}_1\rangle & \cdots & \langle v_0, \tilde{v}_{k-1}\rangle \\ \langle \tilde{v}_1, v_0\rangle & \langle \tilde{v}_1, \tilde{v}_1\rangle & \cdots & \langle \tilde{v}_1, \tilde{v}_{k-1}\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle \tilde{v}_{k-1}, v_0\rangle & \langle \tilde{v}_{k-1}, \tilde{v}_1\rangle & \cdots & \langle \tilde{v}_{k-1}, \tilde{v}_{k-1}\rangle \end{pmatrix}$$

$$= \langle v_0, v_0\rangle \langle \tilde{v}_1, \tilde{v}_1\rangle \cdots \langle \tilde{v}_{k-1}, \tilde{v}_{k-1}\rangle$$

where $\tilde{v}_j$ is the projection of v_j onto $(\mathbf{C}v_0 + \cdots + \mathbf{C}v_{j-1})^{\perp}$, i.e.,

$$\tilde{v}_j = v_j - \sum_{i=0}^{j-1} \frac{\langle v_j, v_i\rangle}{\langle v_i, v_i\rangle} v_i.$$

LEMMA 6.2. *For any n there exist c, $d_0 > 0$ such that for any $d > d_0$ and $\alpha_1, \cdots, \alpha_n$ we have*

$$|t^n - \alpha_1 t^{n-1} - \cdots - \alpha_n| \geqq cd^n \tag{6.3}$$

for at least $d/4$ of the integers $t = 1, 2, \cdots, d$.

Proof. We can write

$$t^n - \alpha_1 t^{n-1} - \cdots - \alpha_n = (t-\gamma_1)\cdots(t-\gamma_n)$$

for some $\gamma_i \in \mathbf{C}$. For each i we have

$$|t-\gamma_i| \geqq |\Re(t-\gamma_i)| \geqq \frac{1}{9n} d$$

for all integers t, $(d/2)-1 \leqq t \leqq d$, except for possibly $(2/9n)d+1$ values of t. If d is sufficiently large we have

$$\frac{2}{9n} d + 1 \leqq \frac{1}{4n} d,$$

and thus for $d/4$ values of t, $(d/2)-1 \leqq t \leqq d$, we have

$$|t^n - \alpha_1 t^{n-1} - \cdots - \alpha_n| \geqq cd^n,$$

where $c = 1/(9n)^n$.

COROLLARY 6.3. *For any k there is a $c > 0$ such that*

$$\langle \tilde{v}_j, \tilde{v}_j\rangle \geqq cm^{2j+1} \tag{6.4}$$

for $j = 0, 1, \cdots, k-1$.

Proof. We have

$$\langle \tilde{v}_j, \tilde{v}_j\rangle = \sum_{i=j}^{d} |i^j - \alpha_1 i^{j-1} - \cdots - \alpha_j|^2 |z_1|^{2(i-j)}$$

for some $\alpha_1, \cdots, \alpha_j$. For sufficiently large d we can estimate this sum as

$$\geqq \sum_{(m/2)-1 \leqq i \leqq m} (ci^j)^2 |z_1|^{2m} \geqq c' m^{2j+1}.$$

Corollary 6.3 establishes (6.2), and thus

$$\psi(0,\cdots,0) \leqq c\left|\prod_{j>i}(z_j-z_i)^2 d^{1+3+\cdots+(2k+1)}\right|^{-1}$$

$$= c\left|\prod_{j>i}(z_j-z_i)^2 d^{k^2}\right|^{-1}.$$

To estimate

$$E\{|F(z_1)|^2\cdots|F(z_k)|^2\}$$

we notice that the equation

$$\frac{f(z_j)-f(z_1)}{z_j-z_1} = f'(z_1)+\frac{z_j-z_1}{2}f''(z_1)+\cdots+\frac{(z_j-z_1)^{k-2}}{(k-1)!}f^{(k-1)}(z_1)$$

$$+\frac{(z_j-z_1)^{k-1}}{k!}f^{(k)}(z_1)(1+O(d^{-\beta}))$$

enables us to write

$$|f'(z_1)-L| \leqq c\left(\prod_{j>1}|z_j-z_1|\right)|f^{(k)}(z_1)|,$$

where L is a linear combination of $f(z_1),\cdots,f(z_n)$. This is true because the linear combination of the above $k-1$ equations that eliminates the $f''(z_1),\cdots,f^{(k-1)}(z_1)$ terms and gives an $f'(z_1)$ term with coefficient 1 on the right-hand side is the linear combination gotten by taking α_j times the z_j equation and adding them, where the α_j's satisfy

$$\begin{pmatrix} 1 & \cdots & 1 \\ \frac{z_2-z_1}{2} & \cdots & \frac{z_k-z_1}{2} \\ \vdots & \ddots & \vdots \\ \frac{(z_2-z_1)^{k-2}}{(k-1)!} & \cdots & \frac{(z_k-z_1)^{k-2}}{(k-1)!} \end{pmatrix}\begin{pmatrix}\alpha_2\\ \alpha_3\\ \vdots\\ \alpha_k\end{pmatrix} = \begin{pmatrix}1\\0\\0\\ \vdots\\0\end{pmatrix}.$$

Using Kramer's rule and solving van der Monde determinants yields

$$\alpha_i = \prod_{j\neq i,1}\frac{z_j-z_1}{z_j-z_i}.$$

The $f^{(k)}(z_1)$ term in the linear combination therefore has coefficient

$$\frac{1}{k!}\sum_{i>1}(z_i-z_1)^{k-1}\prod_{j\neq i,1}\frac{z_j-z_1}{z_j-z_i} = \frac{1}{k!}\left(\prod_{i>1}(z_i-z_1)\right)\sum_{i>1}\frac{(z_i-z_1)^{k-2}}{\prod_{j\neq i,1}(z_j-z_i)}$$

$$= \frac{1}{k!}\left(\prod_{i>1}(z_i-z_1)\right)1,$$

since

$$\sum_{i>1}\frac{(z_i-z_1)^{k-2}}{\prod_{j\neq i,1}(z_j-z_i)} = \frac{\sum_{i>1}[(-1)^{i+1}(z_i-z_1)^{k-2}\prod_{1<n<l,n,l\neq i}(z_l-z_n)]}{\prod_{1<n<l}(z_l-z_n)} = 1,$$

and since the numerator is a polynomial, with the same $z_i^{k-2}\prod_{1<n<l,n,l\neq i}(z_l-z_n)$ coefficient as in $\prod_{1<n<l}(z_l-z_n)$, and the numerator vanishes whenever $z_l=z_n$ for some $l\neq n$.

Hence we may write

$$E\{|F(z_1)|^2 \cdots |F(z_k)|^2\} \leqq c\left(\prod_{j>i} |z_j - z_i|^4\right) E\{|f^{(k)}(z_1)|^2 \cdots |f^{(k)}(z_k)|^2\}$$

where the analogue of Proposition 4.6 for $2k$th powers was used, and by Minkowski's inequality the above is

$$\leqq c\left(\prod_{j>i} |z_j - z_i|^4\right) \sum_{i=1}^{k} E\{|f^{(k)}(z_i)|^{2k}\}$$

$$\leqq c\left(\prod_{j>i} |z_j - z_i|^4\right) m^{k+2k^2},$$

the bound on $E\{|f^{(k)}(z_i)|^{2k}\}$ coming from expanding the expression as in (4.7) and the preceding equation. Combining the above and the estimate on $\psi(0, \cdots, 0)$ yields

$$E\{|f'(z_1)|^2 \cdots |f'(z_k)|^2 \text{ s.t. } f(z_1) = \cdots = f(z_k) = 0\} \leqq c \prod_{j>i} |z_j - z_i|^2 m^{k(k+1)},$$

which gives the desired result.

For the a_i's distributed uniformly, we do the same estimates as in § 5. From the above discussion we see that it suffices to estimate

$$\int_{\mathbf{C}} |t|^{2k} \Upsilon(0, \cdots, 0, t)\, dt,$$

where Υ is the joint density of $\sum a_i u_i, \sum a_i v_i, \cdots, \sum a_i s_i$ with

$$u_i = z_1^i,$$
$$v_i = i z_1^{i-1}(1 + O(d^{-\beta})),$$
$$\vdots$$
$$s_i = i(i-1) \cdots (i-k+1) z_1^{i-k}(1 + O(d^{-\beta})).$$

From here the arguments are just like those in § 5.

COROLLARY 6.4. *The probability that there are k roots within distance* $\delta \leqq d^{-1-\beta}$ *of each other in* $B_2(0) - B_{\delta d^{-1-\beta}}(0)$ *is*

$$\leqq c\delta^{k^2+k-2} d^{k^2+k-1}.$$

Proof. For the proof, integrate Theorem 6.1. □

Applying Corollary 6.4 to the case of $k = 1$ and to some other value of k yields that each root in $B_2(0) - B_{\max(\delta_1, \delta_2) d^{-1-\beta}}(0)$ has no roots within a distance δ_1 and at most $k-1$ roots within a distance δ_2 with probability $\geqq 1 - \tau_1 + \tau_2$, where

$$c\delta_1^4 d^5 = \tau_1$$

and

$$c'\delta_2^{k^2+k-2} d^{k^2+k-1} = \tau_2.$$

If we have at least $d/2$ such roots we get an approximate zero region of area

$$\geqq cd\left(\frac{1}{(k - 1/\delta_1) + (d - k/\delta_2)}\right)^2 \geq cd \min\left(\delta_1^2, \frac{\delta_2^2}{d}\right).$$

Setting $\varepsilon = d\delta_1^2 = \delta_2^2/d$ yields

$$\tau_1 = c\varepsilon^2 d^3, \qquad \tau_2 = cd(\varepsilon d^3)^{(k^2+k-2)/2}.$$

The probability of having $<d/2$ roots in $B_2(0)-B_{\max(\delta_1,\delta_2)d^{-1-\beta}}(0)$ can be estimated as in the proof of Theorem 5.8 and is dominated by $\tau_1+\tau_2$. Hence we can replace the

$$\varepsilon^2 d^7$$

term in (4.2) and (5.1) by

$$\varepsilon^2 d^3+\varepsilon^{(k^2+k-2)/2}d^{(3k^2+3k-4)/2}.$$

We summarize the improved results in the following theorem.

THEOREM 6.5. *For any fixed $N\geqq 2$, there are positive constants c and c' such that*

$$\Pr\{Q_d(\varepsilon)\}\leqq c(\varepsilon^2 d^3+\varepsilon^N d^{3N+1})+2^{-c'd}$$

if the a_i's are normally distributed, and

$$\Pr\{Q_d(\varepsilon)\}\leqq c(\varepsilon^2 d^3+\varepsilon^N d^{3N+1})$$

if the a_i's are uniform.

7. Consequences of the Erdös–Turán estimate. In this section we give two types of improvements of the previous estimates using the following theorem of Erdös and Turán.

LEMMA 7.1. *Let $N(\alpha,\beta)$ be the number of roots of $\sum_{i=0}^d a_i z^i=0$ of the form $re^{i\theta}$ with r, θ real and $\alpha\leqq\theta\leqq\beta$ (and $\beta-\alpha\leqq 2\pi$). Then*

$$\left|N(\alpha,\beta)-\frac{(\beta-\alpha)d}{2\pi}\right|\leqq 16\sqrt{d\log\frac{\sum_{i=0}^d|a_i|}{|a_0||a_d|}}.$$

Proof. For the proof, see [ET50]. □

The following argument assumes that the a_i's are distributed uniformly in $B_1(0)$. The same estimates hold for the a_i's distributed normally, with minor modifications in the arguments. From the above it follows, with probability $\geqq 1-\tau$, that $|a_0|$ is $>c\sqrt{\tau}$ and thus

$$\left|N(\alpha,\beta)-\frac{(\beta-\alpha)d}{2\pi}\right|\leqq c\sqrt{d\log\frac{d}{\sqrt{\tau}}}.$$

This implies that the kth closest root to a given root z has distance

$$\geqq c'\frac{k-c\sqrt{d\log(d/\sqrt{\tau})}}{d}|z|.$$

By Lemma 4.1 we have that more than $d/2$ of the roots lying in $B_\rho(0)$ implies

$$|a_0|\leqq\rho^{d/2}$$

so that

$$\rho\geqq\left(\frac{\tau}{\pi}\right)^{1/d}.$$

It follows that with probability $\geqq 1-\tau-\tau_1-\tau_2$ we have a region of approximate zeros of size

$$\geqq cd\left[\frac{k-1}{\delta_1}+\frac{\sqrt{d\log(d/\sqrt{\tau})}}{\delta_2}+\sum_{k=1}^d\frac{d}{k}\left(\frac{\tau}{\pi}\right)^{-1/d}\right]^{-2}$$

$$\geqq cd\left[\min\left(\delta_1,\delta_2\Big/\sqrt{d\log\frac{d}{\sqrt{\tau}}},\tau^{1/d}/(d\log d)\right)\right]^2$$

where

$$\tau_1 = c'\delta_1^4 d^5, \qquad \tau_2 = c''\delta_2^{N-1} d^N, \qquad N = k^2 + k - 1.$$

Setting

$$\varepsilon = d\delta_1^2 = \delta_2^2 / \log \frac{d}{\sqrt{\tau}},$$

we get

$$\tau_1 = c\varepsilon^2 d^3, \qquad \tau_2 = c\left(\varepsilon \log \frac{d}{\sqrt{\tau}}\right)^{(N-1)/2} d^N.$$

Choosing $\tau = \varepsilon^2 d^2$, and assuming $\varepsilon < 1/d^2$, we get that, with probability,

$$\geqq 1 - c\left(\varepsilon^2 d^3 + \left(\varepsilon \log \frac{1}{\varepsilon}\right)^{(N-1)/2} d^N\right)$$

we have an approximate zero region of area $\geqq c'\varepsilon$, since $\delta_1 = \sqrt{\varepsilon/d}$ is smaller than $\tau^{1/d}/(d \log d) > c\varepsilon^{2/d}/(d \log d)$. Hence we have proven Theorem 1.2, which we restate in the following theorem.

THEOREM 7.2. *For the uniform distribution on $P_d(1)$ we have for any integer N a c such that*

$$\Pr\{Q(\varepsilon)\} < c\left(\varepsilon^2 d^3 + \left(\varepsilon \log \frac{1}{\varepsilon}\right)^{(N-1)/2} d^N\right).$$

Second, we claim that the $\varepsilon^2 d^3$ term can be replaced by $(\varepsilon^2 d^3)^M + e^{-cd}$ for any fixed integer M, c depending on M. This is because Lemma 7.1 gives that with probability $> 1 - e^{-cd}$ we have

$$N(0, \pi/M),\ N(2\pi/M, 3\pi/M), \cdots, N((2M-2)\pi/M, (2M-1)\pi/M)$$

each contains $c'd/M$ roots. For $i = 1, \cdots, M$,

$$z_1^i, \cdots, z_k^i \in N((2i-2)\pi/M, (2i-1)\pi/M) \cap (B_2(0) - B_{\tau^{1/d}}(0)).$$

With $|z_j^i - z_1^i| \leqq c|z_1^i|/d$, we have that

$$P(z_1^1, \cdots, z_k^1, z_1^2, \cdots, z_k^2, \cdots, z_1^M, \cdots, z_k^M) \approx \prod_{i=1}^{M} P(z_1^i, \cdots, z_k^i), \tag{7.1}$$

i.e., the events of finding roots at $z_1^i, \cdots, z_k^i$ for $i = 1, \cdots, M$ are approximately independent. This can be seen by considering for

$$v_j^i \equiv (0, \cdots, 0, j!, \cdots, d(d-1)\cdots(d-j+1)(z_1^i)^{d-j})$$

the matrix

$$\begin{pmatrix} \langle v_0^1, v_0^1\rangle & \langle v_0^1, v_1^1\rangle & \cdots & \langle v_0^1, v_{k-1}^M\rangle \\ \langle v_1^1, v_0^1\rangle & \langle v_1^1, v_1^1\rangle & \cdots & \langle v_1^1, v_{k-1}^M\rangle \\ \vdots & \vdots & \ddots & \vdots \\ \langle v_{k-1}^M, v_0^1\rangle & \langle v_{k-1}^M, v_1^1\rangle & \cdots & \langle v_{k-1}^M, v_{k-1}^M\rangle \end{pmatrix}$$

and noting that its determinant is approximately

$$\prod_{i=1}^{M} \det \begin{pmatrix} \langle v_0^i, v_0^i\rangle & \cdots & \langle v_0^i, v_{k-1}^i\rangle \\ \vdots & \ddots & \vdots \\ \langle v_{k-1}^i, v_0^i\rangle & \cdots & \langle v_{k-1}^i, v_{k-1}^i\rangle \end{pmatrix} = cd^{Mk^2}, \tag{7.2}$$

since for $i \neq j$ the term

$$\langle v_l^i, v_m^j \rangle = O(d^{l+m})$$

instead of proportional to d^{l+m+1}. Hence any term in the expansion of the determinant involving at least one and therefore at least two such terms has size $O(1/d^2)$ times the term in (7.2).

From here on, calculations similar to those done previously yield (7.1) and thus the ability to replace $\varepsilon^2 d^3$ by $(\varepsilon^2 d^3)^M$. Our conclusions are given in Theorem 7.3.

THEOREM 7.3. *For the uniform distribution on $P_d(1)$ we have for any positive integers M and N positive constants c and c' such that*

$$\Pr\{Q(\varepsilon)\} < c\left(e^{-c'd} + (\varepsilon^2 d^3)^M + \left(\varepsilon \log \frac{1}{\varepsilon}\right)^{(N-1)/2} d^N\right).$$

Appendix A. Complex random variables. In this appendix we give some basic facts about complex random variables and their Fourier transforms.

The *real isomorph* of an $m \times n$ complex matrix $M = U + iV$, with U, V real, is the real $2m \times 2n$ matrix given in block form as

$$\hat{M} \text{ or } M^{\wedge} = \begin{pmatrix} U & -V \\ V & U \end{pmatrix}.$$

It is easy to see that $(M_1 + M_2)^{\wedge} = \hat{M}_1 + \hat{M}_2$, $(M_1 M_2)^{\wedge} = \hat{M}_1 \hat{M}_2$, $\hat{M}^{\top} = \hat{M}^*$, where $M^{\top}$ denotes the transpose of M and M^* denotes the complex conjugate transpose of M, and

$$\det \hat{M} = |\det M|^2.$$

For $v = (v_1, \cdots, v_n) \in \mathbf{C}^n$, let

$$\tilde{v} = (\Re(v_1), \cdots, \Re(v_n), \Im(v_1), \cdots, \Im(v_n)) \in \mathbf{R}^{2n}.$$

One can check that $v^{\top} = Au^{\top} \Leftrightarrow \tilde{v}^{\top} = \hat{A}\tilde{u}^{\top}$, and that $\Re\langle u, v\rangle = (\tilde{u}, \tilde{v})$, where $\langle\ ,\rangle$ and $(\ ,\ )$ denote the usual inner products on $\mathbf{C}^n$ and $\mathbf{R}^{2n}$, respectively.

We say that u is a *normally distributed* complex random variable if $\Re(u)$ and $\Im(u)$ are independent, identically and normally distributed, real random variables. The *standard* complex normal u has distribution

$$\phi(z) = \frac{1}{2\pi} e^{-|z|^2/2}.$$

If $w_1, \cdots, w_m$ are independently, normally distributed real random variables with mean 0, and $v_1, \cdots, v_k$ are linear combinations of them, then the $v_1, \cdots, v_k$ have distribution $\phi : \mathbf{R}^k \to \mathbf{R}$

$$\phi(x) = \frac{1}{2\pi^{k/2}} \frac{1}{\sqrt{\det C}} e^{-(C^{-1}x, x)},$$

where C is the variance-covariance matrix for the v's. If $v_j = \sum a_{ij} w_i$, then

$$C = \begin{pmatrix} (a_1, a_1) & (a_1, a_2) & \cdots & (a_1, a_k) \\ (a_2, a_1) & (a_2, a_2) & \cdots & (a_2, a_k) \\ \vdots & \vdots & \ddots & \vdots \\ (a_k, a_1) & (a_k, a_2) & \cdots & (a_k, a_k) \end{pmatrix}$$

where $a_j = (a_{1j}, \cdots, a_{nj})$. Writing $w = Au$, we have $C = AA^{\top}$.

If $u=(u_1,\cdots,u_m)$ are standard complex normals, and $w=Au$, then one has $\tilde{v}^\top=\hat{A}\tilde{u}^\top$. Thus $\tilde{v}^\top$ are real normals with distribution

$$\left(\frac{1}{2\pi}\right)^k \frac{1}{\sqrt{\det C}}\, e^{-(C^{-1}x,x)/2},$$

with $C=\hat{A}\hat{A}^\top$. Hence $C=\hat{B}$, where

$$B=AA^*=\begin{pmatrix}\langle a_1,a_1\rangle & \cdots & \langle a_1,a_k\rangle\\ \vdots & \ddots & \vdots\\ \langle a_k,a_1\rangle & \cdots & \langle a_k,a_k\rangle\end{pmatrix}.$$

Also $C^{-1}=\widehat{B^{-1}}$, and so $(C^{-1}x,x)=\Re\langle B^{-1}z,z\rangle$, where $x=\tilde{z}$. Since $B^{-1}=(AA^*)^{-1}=(A^{-1})^*A^{-1}$, we have $\langle B^{-1}z,z\rangle=\langle A^{-1}z,A^{-1}z\rangle$, which is real, and so $\Re\langle B^{-1}z,z\rangle=\langle B^{-1}z,z\rangle$. Thus w has the distribution

$$\begin{aligned}\phi(z)&=\left(\frac{1}{2\pi}\right)^k \frac{1}{\sqrt{\det C}}\, e^{-(C^{-1}x,x)/2}\\ &=\left(\frac{1}{2\pi}\right)^k \frac{1}{|\det B|}\, e^{-\langle B^{-1}z,z\rangle/2}.\end{aligned}$$

In particular, v_1 and v_2 are independent if and only if $\langle a_1,a_2\rangle=0$.

Next we recall some facts about the characteristic functions (Fourier transform) of complex random variables. For an $\mathbf{R}^n$ valued random variable, u, with density ϕ, its characteristic function $\phi:\mathbf{R}^n\to\mathbf{R}$ is

$$\hat{\phi}(\xi)\equiv E\{e^{i(\xi,u)}\}=\int_{\mathbf{R}^n} e^{i(\xi,x)}\phi(x)\,dx.$$

By the Fourier inversion formula,

$$\phi(x)=\left(\frac{1}{2\pi}\right)^n \int_{\mathbf{R}^n} e^{-i(\xi,x)}\hat{\phi}(\xi)\,d\xi.$$

For a complex valued random variable u with density $\phi:\mathbf{C}\to\mathbf{R}$, we can view u as two real random variables and define its characteristic function

$$\hat{\phi}(\xi_1,\xi_2)=E\{e^{i[\xi_1\Re(u)+\xi_2\Im(u)]}\}=\int_{\mathbf{C}} e^{i\Re\langle\xi,x\rangle}\phi(x)\,dx,$$

where $\xi=\xi_1+i\xi_2$.

If ϕ is radially symmetric, i.e., $\phi(z)=\tilde{\phi}(|z|)$, then we claim that $\hat{\phi}$ is radially symmetric. To see this, note

$$\begin{aligned}\hat{\phi}(\xi)&=\int_{\mathbf{R}^2}\phi(x,y)\,e^{-i(x\xi_1+y\xi_2)}\,dx\,dy\\ &=\int_{r=0}^{\infty}\int_{\theta=0}^{2\pi}\tilde{\phi}(r)\,e^{-ir(\xi_1\cos\theta+\xi_2\sin\theta)}\,d\theta\,r\,dr.\end{aligned}$$

Writing $\xi=|\xi|\,e^{i\psi}$ and substituting this in the above, we get

$$\hat{\phi}(\xi)=\int\!\!\int\tilde{\phi}(r)\,e^{ir|\xi|\cos(\theta-\psi)}\,d\theta\,r\,dr,$$

which is independent of ψ.

Assuming that the fourth moments of u are finite, we have

$$\hat{\phi}(\xi) = \int\int \tilde{\phi}(r)\left(1 + (ir|\xi| \cos\theta) + \frac{(ir|\xi| \cos\theta)^2}{2} + \cdots\right) d\theta\, r\, dr$$
$$= 1 - m_1|\xi|^2 + O(|\xi|^4),$$

since the integrals involving odd powers of $\cos\theta$ vanish. Furthermore, if the $2\ell+2$th moments of u are finite, we have

$$\hat{\phi}(\xi) = 1 - m_1|\xi|^2 - \cdots - m_\ell|\xi|^{2\ell} + O(|\xi|^{2\ell+2}).$$

Furthermore, for any $a \in \mathbf{C}$, it is easy to see that the characteristic function of ua is $\hat{\phi}(|a\xi|^2)$.

The characteristic function of the standard normal v, with density

$$\psi(z) = \frac{1}{2\pi} e^{-|z|^2/2}$$

is

$$\hat{\psi}(\xi) = e^{-|\xi|^2/2} = 1 - \tfrac{1}{2}|\xi|^2 + O(|\xi|^4)$$

for ξ small.

In § 5 we will need to make estimates similar to those used in proving the central limit theorem. For these estimates, we recall the following facts. If u is a complex random variable with density ϕ, then

$$|\hat{\phi}(\xi)| \leqq 1 \qquad \forall \xi.$$

If ϕ is bounded, of bounded support, and, say, has $\phi(x, y)$ for fixed y of bounded total variation in x, then

$$|\hat{\phi}(\xi)| < \frac{a}{|\xi|} \qquad \forall \xi \text{ for some } a.$$

To see this, given ϕ is supported in $[-B, B]^2$, we estimate

$$|\hat{\phi}(\xi)| = \left|\int_{\mathbf{R}^2} \phi(x, y)\, e^{ix|\xi|}\, dx\, dy\right|$$
$$\leqq 2B \max_y \left|\int_{x=-B}^{x=B} \phi(x, y)\, e^{ix|\xi|}\, dx\right|$$
$$= 2B \max_y \left|\phi(x, y) \frac{e^{ix|\xi|}}{i|\xi|}\right|_{-B}^{B} - \int_{-B}^{B} \frac{\partial\phi}{\partial x}(x, y) \frac{e^{ix|\xi|}}{i|\xi|}\, dx\right|$$
$$\leqq \frac{2B}{|\xi|}\left[\max_{x,y} |\phi(x, y)| + \int_{-B}^{B} \left|\frac{\partial\phi}{\partial x}(x, y)\right| dx\right]$$
$$\leqq \frac{c}{|\xi|}\left[\max_{x,y} |\phi(x, y)| + \text{T.V.}_x \phi(x, y)\right].$$

One can calculate the density of $u = u_1 + \cdots + u_n$ by taking the characteristic functions and using

$$\hat{\phi}(\xi) = E\{e^{i\Re\langle\xi, u\rangle}\} = \prod_{j=1}^{n} E\{e^{i\Re\langle\xi, u_i\rangle}\} = \prod_{j=1}^{n} \hat{\phi}_i(\xi).$$

Finally, the following lemma is useful.

LEMMA A.1. *If* $z_1, \cdots, z_n, z_1', \cdots, z_n' \in B_1(0) \subset \mathbf{C}$, *then*

$$|z_1 \cdots z_n - z_1' \cdots z_n'| \leqq \sum_{i=1}^{n} |z_i - z_i'|.$$

Proof. For the proof, see [Bil79]. □

Appendix B. Hammersley's formula. In [Ham60], Hammersley gives the formula

$$P(z_1, \cdots, z_r) = \int_{f(z_1)=\cdots=f(z_r)=0} \frac{|f'(z_1)|^2 \cdots |f'(z_r)|^2}{\prod_{1 \leqq p<q<r} |z_p - z_q|^2} \omega(c)\, dc_r \cdots dc_d, \tag{B.1}$$

where $P(z_1, \cdots, z_r)\, dz_1 \cdots dz_r$ is the probability of finding a root in the region of volume $dz_1 \cdots dz_r$ at $(z_1, \cdots, z_r)$,

$$f(z) = \sum_{i=0}^{d} c_j z^j,$$

and $\omega(c)$ is the density function of the coefficients. Note that this P differs from the P defined in § 4 by a factor of

$$d(d-1) \cdots (d-r+1),$$

since in § 4 we first choose r roots at random to calculate P. Then he claims that

$$P(z_1) = \int_{\mathbf{C}} |t|^2 \psi(0, t)\, dt,$$

where ψ is the joint density of f and f'. More generally we have

$$P(z_1, \cdots, z_r) = E\{|f'(z_1)|^2 \cdots |f'(z_r)|^2 \text{ s.t. } f(z_1) = \cdots = f(z_r) = 0\}$$

where by the right-hand side of the above we mean

$$\int_{\mathbf{C}^r} |t_1|^2 \cdots |t_r|^2 \psi(0, \cdots, 0, t_1, \cdots, t_r)\, dt_1 \cdots dt_r,$$

where ψ is the joint density of $f(z_1), \cdots, f(z_r), f'(z_1), \cdots, f'(z_r)$. This can be derived from (B.1) by setting

$$s_i = \sum_{j=0}^{d} c_j z_i^j$$

$(= f(z_i))$, writing

$$P(z_1, \cdots, z_r) = \lim_{\varepsilon \to 0} \left(\frac{1}{\pi \varepsilon^2}\right)^r \int_{s_i \in B_\varepsilon(0), i=1, \cdots, r} \frac{|f'(z_1)|^2 \cdots |f'(z_r)|^2}{\prod_{1 \leqq p<q<r} |z_p - z_q|^2} \omega(c)\, dc_r \cdots dc_d\, ds_1 \cdots ds_r,$$

and noting that

$$\frac{\partial(s_1, \cdots, s_r, c_r, \cdots, c_d)}{\partial(c_0, \cdots, c_d)} = \frac{\partial(s_1, \cdots, s_r)}{\partial(c_0, \cdots, c_{r-1})} = \prod_{i<j<r} (z_j - z_i).$$

Thus

$$P(z_1, \cdots, z_r) = \lim_{\varepsilon \to 0} \left(\frac{1}{\pi \varepsilon^2}\right)^r \int_{s_i \in B_\varepsilon(0), i=1, \cdots, r} |f'(z_1)|^2 \cdots |f'(z_r)|^2 \omega(c)\, dc_0 \cdots dc_d$$

$$= \int_{\mathbf{C}^r} |t_1|^2 \cdots |t_r|^2 \psi(0, \cdots, 0, t_1, \cdots, t_r)\, dt_1 \cdots dt_r,$$

where ψ is the joint density of $f(z_1), \cdots, f(z_r), f'(z_1), \cdots, f'(z_r)$.

REFERENCES

[Bil79] P. BILLINGSLEY, *Probability and Measure*, John Wiley, New York, 1979.

[BS86] A. T. BHARUCHA-REID AND M. SAMBANDHAM, *Random Polynomials: Probability and Mathematical Statistics*, Academic Press, New York, 1986.

[ET50] P. ERDÖS AND P. TURÁN, *On the distribution of roots of polynomials*, Ann. of Math., 51 (1950), pp. 105–119.

[GT74] W. GREG AND R. TAPIA, *Optimal error bounds for the Newton–Kantorovich theorem*, SIAM J. Numer. Anal., 11 (1974), pp. 10–13.

[Ham60] J. M. HAMMERSLEY, *The zeros of a random polynomial*, in The Third Berkeley Symposium on Mathematical Statistics and Probability, 1960, pp. 89–111.

[KA70] L. KANTOROVICH AND G. AKILOV, *Functional Analysis in Normed Spaces*, Macmillan, New York, 1970.

[Kan52] L. KANTOROVICH, *Functional Analysis and Applied Mathematics*, National Bureau of Standards, Gaithersburg, MD, 1952.

[Kim85] M. KIM, Ph.D. thesis, City University of New York, New York, NY, 1985.

[Mar66] M. MARDEN, *Geometry of Polynomials*, American Mathematical Society, Providence, RI, 1966.

[Ren87] J. RENEGAR, *On the worst-case arithmetic complexity of approximating zeros of polynomials*, preprint, 1987.

[Roy86] H. L. ROYDEN, *Newton's method*, preprint, 1986.

[Sma81] S. SMALE, *The fundamental theorem of algebra and complexity theory*, Bull. Amer. Math. Soc., 4 (1981), pp. 1–35.

[Sma86a] ———, *Algorithms for solving equations*, in The International Congress of Mathematicians, 1986.

[Sma86b] ———, *Newton's method estimates from data at one point*, in The Merging of Disciplines: New Directions in Pure, Applied, and Computational Mathematics, R. E. Ewing, K. I. Gross, and C. F. Martin, eds., Springer-Verlag, Berlin, New York, 1986, pp. 185–196.

[SS85] M. SHUB AND S. SMALE, *Computational complexity: On the geometry of polynomials and a theory of cost: Part* I, Ann. Scient. École Norm. Sup. (4), série *t*, 18 (1985), pp. 107–142.

[SS86] ———, *Computational complexity: On the geometry of polynomials and a theory of cost, Part* II, SIAM J. Comput., 15 (1986), pp. 145–161.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1100-1131, December 1990

010

CATEGORY AND MEASURE IN COMPLEXITY CLASSES*

JACK H. LUTZ†

Abstract. This paper presents *resource-bounded category* and *resource-bounded measure*—two new tools for computational complexity theory—and some applications of these tools to the structure theory of exponential complexity classes.

Resource-bounded category, a complexity-theoretic generalization of the Baire category method, defines nontrivial ideals of *meager* subsets of E, ESPACE, and other complexity classes. Similarly, resource-bounded measure, a generalization of Lebesgue measure theory, defines the *measure* 0 subsets of complexity classes. Properties developed here include a useful characterization of meager sets in terms of *resource-bounded Banach-Mazur games.*

Resource-bounded category and measure are applied to the investigation of uniform versus nonuniform complexity. Kannan's theorem that ESPACE $\not\subseteq$ P/Poly is extended by showing that P/Poly $\cap$ ESPACE is only a meager, measure 0 subset of ESPACE. A theorem of Huynh is extended similarly by showing that all but a meager, measure 0 subset of the languages in ESPACE have high space-bounded Kolmogorov complexity. A new hierarchy of exponential classes is introduced and used to refine known relationships between nonuniform complexity and time complexity.

Known properties of hard languages are also extended. Recent results of Schöning and Huynh state that any language L that is $\leq_m^{\mathrm{P}}$-hard for E or $\leq_T^{\mathrm{P}}$-hard for ESPACE cannot be feasibly approximated. It is proven here that this conclusion in fact holds unless only a meager subset of E is $\leq_m^{\mathrm{P}}$-reducible to L and only a meager, measure 0 subset of ESPACE is $\leq_T^{\mathrm{PSPACE}}$-reducible to L. This suggests a new lower bound method which may be useful in interesting cases.

Key words. resource-bounded category, resource-bounded measure, exponential complexity classes, Banach-Mazur games, Kolmogorov complexity, circuit-size complexity, polynomial reducibilities, hard languages, accessible information, approximable languages

AMS(MOS) subject classifications. 68Q15, 03D15, 68Q30

1. Introduction.

This paper presents *resource-bounded category* and *resource-bounded measure*—two new tools for computational complexity theory—and some applications of these tools to the structure theory of exponential complexity classes.

Like the reducibilities $\leq_T^{\mathrm{P}}$ and $\leq_m^{\mathrm{P}}$ introduced by Cook [1971], Karp [1972], and Levin [1973], and like the generalized Kolmogorov complexities investigated by Hartmanis [1983], Sipser [1983], and others, these tools are complexity-theoretic generalizations of well-developed mathematical methods. Specifically, resource-bounded category generalizes the classical Baire category method and resource-bounded measure generalizes Lebesgue measure theory.

This paper falls naturally into two main parts. In §§ 3-5 we introduce resource-bounded category and measure and their basic properties. In §§ 6-10 we apply these tools to the structural investigation of exponential complexity classes.

Resource-bounded category and measure reveal new structure in certain complexity classes by identifying certain subsets of these classes as "small."

Sets that are small in the sense of category are called *meager.* The *classical* Baire category method (in Oxtoby [1971], for example) says what it means for a subset of a complete metric space to be meager. A computable, or *effective*, version of Baire

* Received by the editors September 8, 1987; accepted for publication (in revised form) April 3, 1990. The work reported here was the author's Ph.D. thesis at the California Institute of Technology and was supported in part by Caltech's Program in Advanced Technologies, sponsored by Aerojet General, General Motors, GTE, and TRW. An earlier version of §§ 3, 4, and 7 was presented at the Second Annual Structure in Complexity Theory Conference, held at Cornell University, Ithaca, New York, June 16-19, 1987.

† Department of Computer Science, Iowa State University, Ames, Iowa 50011.

category was introduced by Mehlhorn [1973] and has also been investigated by Lisagor [1979]. This effective version says what it means for a subset of the set of recursive functions to be meager. The *resource-bounded* version of Baire category developed in § 3 is a natural extension of these ideas that enables us to discuss meager subsets of complexity classes such as E, ESPACE, etc. (See § 2 for notations and terminology used in this introduction.) As it turns out, our formulation is general enough to include classical and effective versions as special cases, so the treatment here is self-contained.

In classical Baire category, meager sets admit a characterization (described in Oxtoby [1971]) in terms of certain two-person infinite games of perfect information, called *Banach–Mazur games. Computable* Banach–Mazur games were introduced in Lisagor [1979] and shown to give an analogous characterization in the effective setting. *Resource-bounded* Banach–Mazur games are introduced in § 4 and shown to (almost) characterize sets that are meager in the corresponding sense.

Suppose a language L is chosen probabilistically by using an independent toss of a fair coin to decide whether each string is in L. Then *classical* Lebesgue measure theory (described in Halmos [1950] and Oxtoby [1971], for example) identifies certain *measurable sets* of languages (also called *events*) and assigns to each measurable set X *a measure* $\mu(X)$, which is the probability that the language so chosen will be an element of X. A set X of languages is then small in the sense of measure if it has measure 0. *Effective* versions of measure theory, which say what it means for a set of computable languages to have measure 0 as a subset of the set of all such languages, have been investigated by Freidzon [1972], Mehlhorn [1974], and others. The *resource-bounded* measure theory introduced in § 5 has the classical and effective theories as special cases, but also defines measurability and measure for subsets of many complexity classes. The small subsets of the complexity class are then the measure 0 sets.

It is tempting to regard the measure of a subset X of a complexity class $\mathscr{C}$ as the "conditional probability" that $L \in X$, given that $L \in \mathscr{C}$, when L is chosen by the above-mentioned experiment. However, this interpretation should not be taken seriously because $\mathscr{C}$ is itself a countable, hence measure 0, subset of the set of all languages. (See the remarks on the Borel paradox in Kolmogorov [1933], for example.)

The main results of §§ 3–5 are the definitions of the resource-bounded meager and measure 0 sets, the justification for calling these sets small (especially Theorems 3.12 and 5.9), the game characterization of meager sets (Theorems 4.3 and 4.4), and a resource-bounded generalization of the classical Kolmogorov zero-one law (Theorem 5.15) stating that measurable sets of interest in complexity theory have measure 0 or 1. Many other results can be proven but are not included here because they are not needed for the applications in the ensuing sections.

The applications in §§ 6–10 all concern the structure of exponential time and space complexity classes. Despite the fact that such classes are far beyond the realm of feasible computation, there are three good reasons for studying their structure.

The first reason is the unfortunate circumstance that many known methods do not work below this level. One of the main areas of complexity theory, the effort to clarify relationships between uniform and nonuniform complexity measures, is currently in this predicament.

A central part of the study of uniform versus nonuniform complexity is the ongoing investigation of (nonuniform, Boolean) circuit-size versus (uniform, algorithmic) time and space. In particular, if P/Poly is the set of languages that have polynomial-size circuits, then it is clear that $\mathrm{P} \subseteq \mathrm{P/Poly}$ and that $\mathrm{P/Poly} \not\subseteq \mathrm{REC}$. The following is also known.

THEOREM 1.1 (Kannan [1982]). $\mathrm{ESPACE} \not\subseteq \mathrm{P/Poly}$.

It is generally believed that $NP \not\subseteq P/Poly$ and in fact Karp and Lipton [1980] have shown that $NP \subseteq P/Poly$ has the unlikely consequence of collapsing the polynomial-time hierarchy to the second level. Nevertheless, the weaker conjectures $NP \not\subseteq SIZE(n)$, $E \not\subseteq SIZE(n)$, and $EXP \not\subseteq P/Poly$ have yet to be proven, and the results of Wilson [1985] show that even these will require nonrelativizable proof techniques.

In § 7 we extend Theorem 1.1 by "widening the separation" between ESPACE and $P/Poly \cap ESPACE$. As a matter of fact, our development of resource-bounded category and measure began with the following question. Among languages in ESPACE, is the phenomenon of not having small (e.g., polynomial-size) circuits rare or is it in some sense typical? In § 7 we show that the phenomenon is very typical in the senses of category and measure. For example, as a subset of ESPACE, $P/Poly \cap ESPACE$ is meager and has measure 0.

Although the results in § 7 were originally proven directly, they are proven here as easy consequences of the results in § 6, where we investigate the relationships between (nonuniform) resource-bounded Kolmogorov complexity and uniform time and space complexity. Our starting point for this is the following known fact, which implies Theorem 1.1.

THEOREM 1.2 (Huynh [1986b]). *There is a language* $L \in ESPACE$ *such that* $KS[2^n](L_{\leq n}) > 2^{n-1}$ *almost everywhere.*

In § 6 we extend this existence theorem by proving an "abundance theorem," which implies that a comeager, measure 1 set of the languages L in ESPACE have $KS[2^n](L_{\leq n}) > 2^n$ infinitely often. It is interesting to note that the category portion of this result is proven by formulating the "voting argument," by which Theorems 1.1 and 1.2 were originally proven, as a winning strategy for a resource-bounded Banach–Mazur game. Moreover, playing this strategy against itself immediately gives Huynh's proof of Theorem 1.2.

In §§ 6 and 7 we also investigate nonuniform complexity in exponential time classes, but the results here are less satisfying. As mentioned earlier, an analogue of Theorem 1.1 for E is conjectured but will probably be very hard to prove. The same holds for Theorem 1.2. Nevertheless, we can generalize the notion of exponential time to more accurately pinpoint the limits of relativizable methods, and then prove category and measure results right up to these limits.

To this end, we introduce the G-hierarchy $G_0, G_1, \cdots$ in § 2. Each G_i is a class of functions from $\mathbf{N}$ to $\mathbf{N}$, these functions being regarded as growth rates. The class G_0 contains all linearly bounded growth rates and the class G_1 contains all polynomially bounded growth rates. Each class G_i is closed under composition and each class G_{i+1} contains growth rates which asymptotically dominate all growth rates in G_i. Thus, for $i > 1$, G_i contains superpolynomial growth rates. Nevertheless, every element of $\bigcup_{i=0}^{\infty} G_i$ is $o(2^n)$, i.e., subexponential.

We then define a hierarchy $E_1, E_2, \cdots$ via $E_i = DTIME(2^{G_{i-1}})$. The first two levels of this hierarchy are the widely studied exponential time complexity classes $E_1 = DTIME(2^{linear}) = E$ and $E_2 = DTIME(2^{polynomial}) = EXP$. Here we use the expression "exponential time complexity class" to refer to any of the classes E_i. In §§ 6 and 7 we investigate nonuniform complexity in these classes. Among other things, we show that $P/Poly$ is meager and has measure 0 in E_3 and that $SIZE(n^k)$ is meager and has measure 0 in E_2. Since Wilson [1985] has exhibited oracles relative to which $E_2 \subseteq P/Poly$ and $E \subseteq SIZE(n)$, these results are essentially the strongest that can be proven by relativizable means.

The second reason for studying the structure of exponential complexity classes is the recent emergence of results that relate questions about these classes to open

questions about classes at lower levels. As just one example, we note that Hartmanis and Yesha [1984] have shown that P/Poly $\cap$ PSPACE $\neq$ P if and only if E $\neq$ ESPACE.

The third reason for studying the structure of exponential complexity classes is derivative from the first, and is the motivation for §§ 8–10 of this paper. This is the fact that, unlike lower-level classes, the exponential classes have been proven to contain intractable problems. For the purpose of proving intractability of specific problems—arguably the most important objective of complexity theory—this existence of intractability is a valuable resource.

In practice, proofs that specific languages L are intractable have taken the following three-part form:

(i) A complexity class $\mathscr{C}$ is shown by diagonalization to contain an intractable language. (The language so constructed does not correspond to any natural problem.)

(ii) The specific language L is then shown to be polynomial-time *hard* for $\mathscr{C}$, i.e., it is shown that every language in $\mathscr{C}$ is polynomial-time reducible to L.

(iii) It is inferred from (i) and (ii) that some intractable problem is reducible to L, whence L itself must be intractable.

Thus the structure of the class $\mathscr{C}$ under polynomial-time reducibility allows us to infer the intractability of a specific problem from the existence of intractability in $\mathscr{C}$.

The advantage of this method is that part (ii), a "positive" assertion about what *can* be efficiently computed, is easier to establish by known methods than a direct proof of the "negative" assertion that L *cannot* be efficiently computed.

For example, a number of problems are now known to be intractable because they are polynomial-time hard for E or ESPACE. (For examples, see Meyer and Stockmeyer [1972] and Stockmeyer and Chandra [1979].) Similarly, the real significance of the P versus NP question is the fact that many extremely important computational problems are known to be hard for NP, so a proof by any means that P $\neq$ NP will imply immediately that these problems are not in P.

The properties of languages that are hard for various complexity classes have been investigated extensively. Recently it has been shown that the intractability of hard languages for E and ESPACE also includes lower bounds for "approximate recognition." In particular, the following two facts are known.

THEOREM 1.3 (Schöning [1986], Huynh [1986a]). *If L is $\leqq_m^{\mathrm{P}}$-hard for* E, *then L is 2^{n^c} far from* P *for some $c > 0$.*

THEOREM 1.4 (Huynh [1986b]). *if L is $\leqq_T^{\mathrm{P}}$-hard for* ESPACE, *then L is 2^{n^c} far from* P *for some $c > 0$.*

Unfortunately, most of the problems that we would like to prove intractable are probably not hard for such large classes as E or ESPACE. Efforts to prove the intractability of these problems have thus focused on carrying out part (i) of the above method for smaller classes $\mathscr{C}$.

Here we propose a different remedy. Let $\mathscr{R}(L)$ be the set of languages that are polynomial-time reducible to L. Part (ii) of the above method requires us to show that $\mathscr{C} \subseteq \mathscr{R}(L)$, i.e., that L contains *all* information about $\mathscr{C}$ in $\mathscr{R}$-accessible form. In §§ 9 and 10 we prove that, for $\mathscr{C} = \mathrm{E}$ and $\mathscr{C} = \mathrm{ESPACE}$, it suffices just to prove that $\mathscr{R}(L) \cap \mathscr{C}$ is a nonsmall subset of $\mathscr{C}$, i.e., that L contains a "substantial amount" of information about $\mathscr{C}$. Specifically, we prove that the conclusion of Theorem 1.3 holds if a nonmeager subset of the languages in E are $\leqq_m^{\mathrm{P}}$-reducible to L. Similarly, the conclusion of Theorem 1.4 holds if either a nonmeager or a nonmeasure 0 subset of the languages in ESPACE is $\leqq_T^{\mathrm{PSPACE}}$-reducible to L. Stated in the contrapositive, these results say that any language that is feasibly approximable contains very little accessible information about the class $\mathscr{C}$. In the course of proving these results we also prove that "most" languages in E and ESPACE are intractable, even to approximation.

Although it appears that we have greatly weakened the hypotheses of Theorems 1.3 and 1.4, this has not been proven. It is conceivable that every language that contains nonmeager or nonmeasure 0 accessible information about one of these classes is actually hard for that class. In § 8, after introducing some notation, we formulate some *partial information hypotheses.* These are conjectures that assert the existence of languages $L \in \mathscr{C}$ such that $\mathscr{R}(L) \cap \mathscr{C}$ is not small in $\mathscr{C}$ and also does not equal $\mathscr{C}$, i.e., L contains "accessible partial information" about $\mathscr{C}$. If these conjectures hold, and can be proven, then the methods provided by §§ 9 and 10 may lead to interesting intractability proofs.

The main results of §§ 6–10 are the extensions of Theorems 1.1, 1.2, 1.3, and 1.4 (Theorems 7.2, 6.1, 10.6, and 9.3, respectively), the analysis of nonuniform complexity versus exponential time (Theorems 6.6 and 7.6), and the fact that "most" languages in E and ESPACE are hard to approximate (Theorems 9.1 and 10.5).

It should be noted here that the very interesting "highness" and "lowness" properties investigated by Schöning [1983]; Balcázar, Book, and Schöning [1986]; and others are somewhat analogous to the notions of accessible information content introduced in § 8.

2. Preliminaries. If X and Y are sets, then $X \setminus Y = \{x \in X \mid x \notin Y\}$, $X \triangle Y = (X \setminus Y) \cup (Y \setminus X)$ is the symmetric difference of X and Y, and $|X|$ is the cardinality of X.

We write $\{0,1\}^*$ for the set of all finite binary strings, $|x|$ for the length of a string x, λ for the empty string, $\{0,1\}^+$ for $\{0,1\}^* \setminus \{\lambda\}$, $\{0,1\}^{\leqq n}$ for $\{x \in \{0,1\}^* \mid |x| \leqq n\}$, and $\{0,1\}^n$ for $\{x \in \{0,1\}^* \mid |x| = n\}$. We fix the lexicographic enumeration $s_0 = \lambda$, $s_1 = 0$, $s_2 = 1$, $s_3 = 00, \cdots$ of $\{0,1\}^*$ and let "next" be the successor function for this enumeration, i.e., $\text{next}(s_k) = s_{k+1}$. We write $x \sqsubseteq y$ to indicate that x is an initial substring of y, i.e., $y = xz$ for some z.

All functions, unless otherwise stated, are from $\{0,1\}^*$ into $\{0,1\}^*$. Such functions are also called *transductions.* We write f^n for the n-fold composition of f with itself.

We say a condition $\Theta(n)$ holds *almost everywhere* if it holds for all but finitely many $n \in \mathbf{N}$. We say $\Theta(n)$ holds *infinitely often* if it holds for infinitely many $n \in \mathbf{N}$.

All *languages* here are sets $L \subseteq \{0,1\}^*$; we write $\mathscr{P}(\{0,1\}^*)$ for the set of all languages. We identify a language L with its *characteristic bitstring* $b_0 b_1 b_2, \cdots$, where b_k is 1 if $s_k \in L$ and 0 otherwise. A string x is an *initial bitmap* of a language L, and we write $x \sqsubseteq L$, if x is an initial substring of the characteristic bitstring of L. We write $L_{\leqq n}$ for $L \cap \{0,1\}^{\leqq n}$ and $L_{=n}$ for $L \cap \{0,1\}^n$.

Our model of algorithmic computation is the multitape Turing machine (TM). We write REC for the set of languages L that can be recognized by a deterministic TM. For a resource bound $t: \mathbf{N} \to \mathbf{N}$ we write DTIME (t) (respectively, DSPACE (t)) for the set of languages L that can be decided by a deterministic TM that halts in $O(t(n))$ steps (respectively, after using $O(t(n))$ workspace) on inputs of length n. Similarly, NTIME (t) is the set of languages L that can be accepted by a nondeterministic TM that halts in $O(t(n))$ steps on all accepting computations. We mention the following well-known uniform complexity classes:

$$\mathrm{P} = \cup\{\mathrm{DTIME}(n^k) \mid k \in \mathbf{N}\},$$
$$\mathrm{NP} = \cup\{\mathrm{NTIME}(n^k) \mid k \in \mathbf{N}\},$$
$$\mathrm{PSPACE} = \cup\{\mathrm{DSPACE}(n^k) \mid k \in \mathbf{N}\},$$
$$\mathrm{E} = \cup\{\mathrm{DTIME}(2^{kn}) \mid k \in \mathbf{N}\},$$
$$\mathrm{ESPACE} = \cup\{\mathrm{DSPACE}(2^{kn}) \mid k \in \mathbf{N}\},$$
$$\mathrm{EXP} = \cup\{\mathrm{DTIME}(2^{n^k}) \mid k \in \mathbf{N}\},$$

$$\text{EXPSPACE} = \cup\{\text{DSPACE}(2^{n^k}) \mid k \in \mathbf{N}\}.$$

For each $i \in \mathbf{N}$ we define a class G_i of functions from $\mathbf{N}$ into $\mathbf{N}$ as follows.

$$G_0 = \{f \mid (\exists k) f(n) \leqq kn \text{ almost everywhere}\}$$

$$G_{i+1} = 2^{G_i(\log n)} = \{f \mid (\exists g \in G_i) f(n) \leqq 2^{g(\log n)} \text{ almost everywhere}\}$$

(The logarithm here is $\log n = \min\{l \in \mathbf{N} \mid 2^l \geqq n\}$.) For $i \in \mathbf{N}$, we define the function $\hat{g}_i \in G_i$ by $\hat{g}_0(n) = 2n$, $\hat{g}_{i+1}(n) = 2^{\hat{g}_i(\log n)}$. We will think of the functions in these classes as growth rates. In particular, G_0 contains the linearly bounded growth rates and G_1 contains the polynomially bounded growth rates. It is easy to prove by induction that for each $i \in \mathbf{N}$, the following hold:

(i) G_i is closed under composition.

(ii) For each $f \in G_i$, $f(n) = o(\hat{g}_{i+1}(n))$.

(iii) $\hat{g}_i(n) = o(2^n)$.

(For example, consider the induction step in (iii). Assuming $\hat{g}_i(n) = o(2^n)$, we have $\log(\hat{g}_{i+1}(n)) = o(n)$, so $\log(\hat{g}_{i+1}(n)/2^n) \to -\infty$, so $\hat{g}_{i+1}(n) = o(2^n)$.) Thus, for each $i > 1$, G_i contains superpolynomial growth rates, but all growth rates in the G-hierarchy are subexponential.

Using the G-hierarchy, we define, for $i \geqq 1$, the following uniform complexity classes:

$$E_i = \text{DTIME}(2^{G_{i-1}}) = \cup\{\text{DTIME}(2^g) \mid g \in G_{i-1}\},$$

$$E_i\text{SPACE} = \text{DSPACE}(2^{G_{i-1}}).$$

Using the standard hierarchy theorems for deterministic time and space, it is clear that $E_i \subsetneqq E_{i+1}$ and $E_i\text{SPACE} \subsetneqq E_{i+1}\text{SPACE}$ for each $i \geqq 1$. It is also clear that $E_1 = E$, $E_2 = \text{EXP}$, $E_1\text{SPACE} = \text{ESPACE}$, and $E_2\text{SPACE} = \text{EXPSPACE}$, i.e., the first two levels of these hierarchies are the well-known exponential complexity classes. We generalize this terminology by calling each E_i an exponential time complexity class and each $E_i\text{SPACE}$ an exponential space complexity class. (Note that all these are well below doubly exponential levels.) The class $E_3 = \text{DTIME}(2^{n^{(\log n)^{O(1)}}})$ will be of particular interest.

We also define some classes of transductions, i.e., of functions that transform strings. The computational model we use for this is the TM transducer, which is a TM augmented with a write-only output tape, the contents of which are not counted as workspace. To avoid confusing transduction classes with complexity classes of languages, we will write transduction classes using lowercase letters. The following classes are used:

$$\text{all} = \{f \mid f\colon \{0,1\}^* \to \{0,1\}^*\},$$

$$\text{rec} = \{f \in \text{all} \mid f \text{ is recursive}\},$$

$$p_i = \{f \in \text{all} \mid f \text{ is computable in } G_i \text{ time}\},$$

$$p_i\text{space} = \{f \in \text{all} \mid f \text{ is computable in } G_i \text{ space}\}.$$

We write p and pspace for p_1 and $p_1$space, respectively.

If L_1 and L_2 are languages, then a *many-one reduction* of L_1 to L_2 is a transduction g such that for all $x \in \{0,1\}^*$, $x \in L_1$ if and only if $g(x) \in L_2$. As usual, then, L_1 is *polynomial-time many-one reducible* to L_2, and we write $L_1 \leqq_m^{\text{P}} L_2$, if some $g \in \text{p}$ is a many-one reduction of L_1 to L_2.

For Turing reducibilities we use oracle machines. An oracle machine is a TM M augmented with a write-only oracle tape. An arbitrary language $A \subseteq \{0,1\}^*$ may be used as the oracle. During any cycle of execution, the machine is allowed to "query the oracle," i.e., to base its next action on whether or not the string currently written on the oracle tape is an element of A. We write $L(M^A)$ to denote the language decided by M when equipped with the oracle A. We then say L_1 is *polynomial-time Turing reducible* to L_2, and write $L_1 \leqq_T^{\mathrm{P}} L_2$, if there is a polynomial-time-bounded oracle machine M such that $L_1 = L(M^{L_2})$. The *polynomial-space Turing reducibility* $\leqq_T^{\mathrm{PSPACE}}$ is defined analogously, with the convention that the oracle tape is counted as workspace. For $i \geqq 1$, we also consider the Turing reducibility $\leqq_T^{\mathrm{P}_i}$, which is like $\leqq_T^{\mathrm{P}}$, except that the oracle machine is G_i-time-bounded.

For a fixed TM M, a resource bound t, a language L, and $n \in \mathbf{N}$, the *t-time-bounded Kolmogorov complexity* of $L_{\leqq n}$ relative to M is

$$KT_M[t](L_{\leqq n}) = |\pi|,$$

where π is the shortest "program," i.e., binary string, such that for each $x \in \{0,1\}^{\leqq n}$, the machine M on input $\langle \pi, x \rangle$ correctly decides in $\leqq t$ steps whether $x \in L$. Since $\{0,1\}^{\leqq n} = \{s_0, \cdots, s_{2^{n+1}-2}\}$, we abbreviate this condition by saying that $M(\langle \pi, s_i \rangle)_{i=0}^{2^{n+1}-2} = L_{\leqq n}$ in $\leqq t$ time. If there is no such program, then $KT_M[t](L_{\leqq n}) = \infty$. The *space-bounded Kolmogorov complexity* $KS_M[t](L_{\leqq n})$ is defined similarly, except that M decides membership in L using $\leqq t$ cells of worktape.

It is well known (see Huynh [1986b], for example) that there exist a universal TM U and a polynomial p such that for each TM M there is a constant c such that the following hold for all t, L, and n:

(i) $KT_U[t](L_{\leqq n}) \leqq KT_M[p(ct+c)](L_{\leqq n}) + c$.

(ii) $KS_U[t](L_{\leqq n}) \leqq KS_M[ct+c](L_{\leqq n}) + c$.

As usual, we fix such a universal machine U and omit it from the notation. The time- and space-bounded Kolmogorov complexities $KT[t](L_{=n})$ and $KS[t](L_{=n})$ are defined analogously.

All *circuits* here are Boolean (combinational, acyclic) circuits over the basis {and, or, not, 0, 1}. (All gates have fan-in $\leqq 2$; fan-out is unbounded.) A circuit has some number n of inputs and a distinguished output gate, at which it computes a set $S \subseteq \{0,1\}^n$ in the usual way. (The set S consists of those input strings for which the circuit's output is 1.) The *size* of a circuit c is the number size(c) of gates in c. (Inputs are not gates.) The *circuit-size complexity* of a language L is the function $CS_L : \mathbf{N} \to \mathbf{N}$ defined by

$$CS_L(n) = \min\{\mathrm{size}(c) \mid c \text{ computes } L_{=n}\}.$$

We will insist that a *circuit-size bound* be a function $f : \mathbf{N} \to \mathbf{N}$, which is computable in space polynomial in n, nowhere zero, and such that $\lim_{n\to\infty} n2^{-n} f(n)$ exists (or is infinite). For each such f we define the nonuniform complexity class

$$\mathrm{SIZE}(f) = \{L \subseteq \{0,1\}^* \mid CS_L = O(f)\}.$$

We call f *trivial* if $\mathrm{SIZE}(f) = \mathscr{P}(\{0,1\}^*)$; otherwise it is *nontrivial*. Well-known theorems of Shannon [1949] and Lupanov [1958] establish that a circuit-size bound f is nontrivial if and only if $\lim_{n\to\infty} n2^{-n} f(n) = 0$. For any circuit-size bound f, we note that $\mathrm{SIZE}(f)$ has the cardinality of the continuum and hence contains many nonrecursive languages. Finally, we define the set $\mathrm{P/Poly} = \cup\{\mathrm{SIZE}(n^k) \mid k \in \mathbf{N}\}$ of all languages that have polynomial-size circuits.

Following Yesha [1983], Schöning [1986], and others, a language L is $f(n)$ *close* to a complexity class $\mathscr{C}$ if there is a language $L' \in \mathscr{C}$ such that $|(L \triangle L')_{\leqq n}| < f(n)$ almost everywhere. Otherwise, L is $f(n)$ *far* from $\mathscr{C}$.

3. Resource-bounded Baire category. In this section we introduce and develop a general formulation of notions of Baire category on $\mathcal{P}(\{0,1\}^*)$. The formulation is general enough to admit the classical and effective notions as special cases, but its real significance is that it admits *resource-bounded* versions of Baire category, which will be of use in computational complexity theory.

We will define a notion of category to be a class of functions from $\{0,1\}^*$ to $\{0,1\}^*$ that contains certain initial functions and is closed under certain operations. Our first task is thus to specify these initial functions and operations.

We fix once and for all a one-to-one *pairing function* $\langle , \rangle$ from $\{0,1\}^*$ onto $\{0,1\}^*$ such that the pairing function and its associated *projections* $\pi_1(\langle x, y\rangle) = x$, $\pi_2(\langle x, y\rangle) = y$ are computable in polynomial time. We insist further that $\langle x, y\rangle \in \{0\}^*$ if and only if $x, y \in \{0\}^*$. This latter condition canonically induces a pairing function $\langle , \rangle$ from $\mathbf{N} \times \mathbf{N}$ onto $\mathbf{N}$. We write $\langle x, y, z\rangle$ for $\langle x, \langle y, z\rangle\rangle$, etc., so that tuples of any fixed length are coded by the pairing function.

By the *conditional function* we mean the function cond $(\langle x, y, z, w\rangle)$ whose value is z if $x \sqsubseteq y$, and w otherwise.

The *composition* $f \circ g$, *concatenation* fg, and *pairing* $\langle f, g\rangle$ of two functions f and g are defined by $(f \circ g)(x) = f(g(x))$, $(fg)(x) = f(x)g(x)$, and $\langle f, g\rangle(x) = \langle f(x), g(x)\rangle$, respectively.

The functions and operations defined thus far are natural and somewhat standard in the theory of subrecursive function classes. In the following definition, two more necessary operations are specified. These operations are a little more awkward to state than the preceding ones but are natural in the context of resource-bounded computation. Both are called "inversion" operations because they involve reconstructing ways in which a string could have been built up by recursion.

For a function f and $k \in \mathbf{N}$ it will be convenient to define the function $f_k(x) = f(\langle 0^k, x\rangle)$. The function f can thus be considered a "uniform enumeration" of the functions $f_0, f_1, \cdots$.

DEFINITION 3.1. Let f be a function.

(1) The *type* I *inversion* of f is $f^{\mathrm{I}}(x) = \langle 0^k, x_k\rangle$, where k is maximum such that the sequence $x_0 = \lambda$, $x_{n+1} = f_n(x_n)$ satisfies

$$x_0 \underset{\neq}{\sqsubseteq} x_1 \underset{\neq}{\sqsubseteq} \cdots \underset{\neq}{\sqsubseteq} x_k \sqsubseteq x.$$

(2) The *type* II *inversion* of f is $f^{\mathrm{II}}(x) = 0^m$, where $m = 0$ if $x = \lambda$ and otherwise m is maximum such that there exists a sequence $x_0, \cdots, x_{m-1}$ satisfying

$$\lambda \underset{\neq}{\sqsubseteq} x_0 \underset{\neq}{\sqsubseteq} f_0(x_0) \underset{\neq}{\sqsubseteq} x_1 \underset{\neq}{\sqsubseteq} \cdots \underset{\neq}{\sqsubseteq} x_{m-1} \underset{\neq}{\sqsubseteq} f_{m-1}(x_{m-1}) \underset{\neq}{\sqsubseteq} x.$$

Since $f^{\mathrm{I}}(x)$ and $f^{\mathrm{II}}(x)$ specify "internal" properties of the string x with respect to f, it is also natural to think of these inversion operations as "internal primitive recursion" and "internal search recursion," respectively.

The general formulation of notions of Baire category can now be stated and developed.

DEFINITION 3.2. A *notion of category* is a class Γ of functions from $\{0,1\}^*$ into $\{0,1\}^*$ which contains the projections, all constant functions, and the conditional function, and which is closed under composition, concatenation, pairing, and type I inversion. A notion of category Γ has the *Mazur property* if it is also closed under type II inversion.

From now on, Γ, Γ', etc. will denote notions of category.

The essential link between Definition 3.2 and the development of a Baire category theory for $\mathcal{P}(\{0,1\}^*)$ is the simple observation that binary languages can be constructed by using functions from $\{0,1\}^*$ into $\{0,1\}^*$.

A *constructor* is a function γ which satisfies $x \underset{\neq}{\sqsubseteq} \gamma(x)$ for all $x \in \{0,1\}^*$. The *result* of a constructor γ (i.e., the language *constructed* by γ) is the unique language $R(\gamma)$ such that $\gamma^k(\lambda) \sqsubseteq R(\gamma)$ for every $k \in \mathbf{N}$. A *clocked constructor* is a function γ which satisfies $x \underset{\neq}{\sqsubseteq} \gamma(\langle w, x\rangle)$ for all $w \in \{0\}^*$ and $x \in \{0,1\}^*$. The *result* of a clocked constructor γ is the unique language $R(\gamma)$ such that $x_k \sqsubseteq R(\gamma)$ for every $k \in \mathbf{N}$, where $x_0 = \lambda$ and $x_{k+1} = \gamma_k(x_k)$. (No confusion will result from the obvious ambiguity here.)

Intuitively, a constructor or clocked constructor γ builds a language $R(\gamma)$ by starting with λ and then iteratively generating successively longer initial bitmaps of $R(\gamma)$. A clocked constructor is permitted to have an "agenda that varies over time," i.e., to extend the initial bitmap in a way which depends upon the stage k of the construction.

For each Γ, Γ_c denotes the set of all constructors in Γ and Γ_{cc} denotes the set of all clocked constructors in Γ. It is then natural to define the sets $R(\Gamma_c)$, $R(\Gamma_{cc})$ of all $R(\gamma)$ such that $\gamma \in \Gamma_c$, $\gamma \in \Gamma_{cc}$, respectively.

LEMMA 3.3. *If Γ is a notion of category, then $R(\Gamma_c) = R(\Gamma_{cc})$.*

Proof. If $\gamma \in \Gamma_c$, then $\gamma \circ \pi_2 \in \Gamma_{cc}$ and $R(\gamma \circ \pi_2) = R(\gamma)$, so $R(\gamma) \in R(\Gamma_{cc})$. Conversely, let $\gamma \in \Gamma_{cc}$ and define $x_0 = \lambda$, $x_{n+1} = \gamma_n(x_n)$. It is clear that

(i) $\gamma^1(x_n) = \langle 0^n, x_n\rangle$ for every $n \in \mathbf{N}$.

Also, by the maximality of k in the definition of γ^1, we have

(ii) $\gamma(\gamma^1(x)) \not\sqsubseteq x$ for every $x \in \{0,1\}^*$.

Define

$$\hat{\gamma}(x) = \begin{cases} \gamma(\gamma^1(x)) & \text{if } x \sqsubseteq \gamma(\gamma^1(x)) \\ x0 & \text{otherwise.} \end{cases}$$

Since γ is a notion of category and $\gamma \in \Gamma$, it is clear that $\hat{\gamma} \in \Gamma$. Also, it is clear from (ii) that $x \underset{\neq}{\sqsubseteq} \hat{\gamma}(x)$ for all $x \in \{0,1\}^*$, whence $\hat{\gamma} \in \Gamma_c$.

We now show by induction that $\hat{\gamma}^n(\lambda) = x_n$ for all $n \in \mathbf{N}$. For $n = 0$ we have $\hat{\gamma}^0(\lambda) = \lambda = x_0$, so this is clear. For the induction step, first note that $x_n \sqsubseteq \gamma(\langle 0^n, x_n\rangle) = \gamma(\gamma^1(x_n))$ by (i) and the fact that γ is a clocked constructor. It follows by the induction hypothesis, the definition of $\hat{\gamma}$, and (i) that

$$\hat{\gamma}^{n+1}(\lambda) = \hat{\gamma}(x_n) = \gamma(\gamma^1(x_n)) = \gamma(\langle 0^n, x_n\rangle) = \gamma_n(x_n) = x_{n+1},$$

whence the induction is complete.

Since $\hat{\gamma}^n(\lambda) = x_n$ for all $n \in \mathbf{N}$, we must have $R(\gamma) = R(\hat{\gamma}) \in R(\Gamma_c)$. □

Lemma 3.3 says that it makes no difference whether constructors or clocked constructors are used in a notion of category. This justifies the following definition.

DEFINITION 3.4. The *result class* of Γ is $R(\Gamma) = R(\Gamma_c) = R(\Gamma_{cc})$.

The following routine lemma is the reason for our interest in the transduction classes defined in § 2.

LEMMA 3.5. *The classes* all, rec, p_i $(i \geqq 1)$, *and* $\mathrm{p}_i$space $(i \geqq 1)$ *are notions of category with result classes as follows.*

(1) $R(\mathrm{all}) = \mathcal{P}(\{0,1\}^*)$.

(2) $R(\mathrm{rec}) = \mathrm{REC}$.

(3) $R(\mathrm{p}_i) = \mathrm{E}_i$.

(4) $R(\mathrm{p}_i\mathrm{space}) = \mathrm{E}_i\mathrm{SPACE}$.

The classes all, rec, *and* $\mathrm{p}_i$space $(i \geqq 1)$ *have the Mazur property.*

Proof. We prove (3). The proofs of (1), (2), and (4) are similar.

Let $L = R(\gamma)$, where $\gamma \in \mathrm{p}_i$ is a constructor. Consider a TM M which does the following:

begin
  input x;
  $n :=$ the unique index such that $s_n = x$;
  $z := \lambda$;
  while $|z| \leqq n$ **do** $z := \gamma(z)$;
  if the nth bit of z (counting from 0) is 1
    then accept
    else reject
end M.

(Recall that $s_0, s_1, s_2, \cdots$ is the standard enumeration of $\{0,1\}^*$. The computation of n is very simple because $1x$ is the usual binary representation of $n+1$.) It is clear that M recognizes the language $R(\gamma) = L$. Since $\gamma \in \mathrm{p}_i$, the assignment $z := \gamma(z)$ takes $\mathrm{G}_i(|z|)$ time, so the while loop takes at most

$$(n+1)\mathrm{G}_i(n) = 2^{O(|x|)}\mathrm{G}_i(2^{O(|x|)}) = \mathrm{G}_i(2^{|x|}) = 2^{\mathrm{G}_{i-1}(|x|)}$$

time. Thus $L \in \mathrm{DTIME}\,(2^{\mathrm{G}_{i-1}}) = \mathrm{E}_i$.

Conversely, let $L \in \mathrm{E}_i$. Then there is a $2^{\mathrm{G}_{i-1}}$-time bounded TM M that recognizes L. Let γ be the function computed by the following algorithm.

begin
  input z;
  if M accepts $s_{|z|}$
    then output $z1$
    else output $z0$
end γ.

It is clear that γ is a constructor with $R(\gamma) = L$. In fact, $\gamma \in \mathrm{p}_i$, since the simulation of M on input $s_{|z|}$ takes at most

$$2^{\mathrm{G}_{i-1}(|s_{|z|}|)} = \mathrm{G}_i(2^{|s_{|z|}|}) = \mathrm{G}_i(|z|)$$

time. Thus $L = R(\delta) \in R(\mathrm{p}_i)$. □

The classes all and rec are, respectively, the *classical* and *effective* notions of category on $\mathscr{P}(\{0,1\}^*)$. The classes p_i and $\mathrm{p}_i$space are *resource-bounded* notions of category. Of course many other such notions can be defined, e.g., by relativization, variation of the resources or bounds, etc.

We conjecture that the time-bounded classes p_i do not have the Mazur property, but this is probably hard to prove, since it implies, for example, that p does not contain every function that is computable on-line in polynomial space.

The significance of a notion of category Γ lies in the structure it imposes on $\mathscr{P}(\{0,1\}^*)$ and on the result class $R(\Gamma)$. In particular, these structures yield natural notions of "smallness" for subsets of $\mathscr{P}(\{0,1\}^*)$ and $R(\Gamma)$. The structures on $\mathscr{P}(\{0,1\}^*)$ and on $R(\Gamma)$ are analogous and will be developed in parallel.

DEFINITION 3.6. For each $x \in \{0,1\}^*$, the *basic set* about x in $\mathscr{P}(\{0,1\}^*)$ is $B_x = \{L \subseteq \{0,1\}^* \mid x \sqsubseteq L\}$. The corresponding *basic set* about x in $R(\Gamma)$ is $B_x \cap R(\Gamma)$.

DEFINITION 3.7. A set $Z \subseteq \mathscr{P}(\{0,1\}^*)$ is Γ-*nowhere dense* (respectively, *nowhere dense in* $R(\Gamma)$) if there exists $h \in \Gamma_c$ such that $B_{h(x)} \cap Z = \varnothing$ (respectively, $B_{h(x)} \cap R(\Gamma) \cap Z = \varnothing$) holds for every $x \in \{0,1\}^*$.

Intuitively, a set Z is Γ-nowhere dense if Γ provides sufficient resources to compute from any basic set B_x a basic set $B_y \subsetneq B_x$ which completely avoids Z.

The following lemma summarizes some easy properties of nowhere dense sets.

LEMMA 3.8. (1) *If Z is Γ-nowhere dense, then Z is nowhere dense in $R(\Gamma)$.*

(2) *If Z is Γ-nowhere dense and $\Gamma \subseteq \Gamma'$, then Z is Γ'-nowhere dense.*

(3a) *Subsets of Γ-nowhere dense sets are Γ-nowhere dense.*

(3b) *Subsets of sets nowhere dense in $R(\Gamma)$ are nowhere dense in $R(\Gamma)$.*

(4a) *Finite unions of Γ-nowhere dense sets are Γ-nowhere dense.*

(4b) *Finite unions of sets nowhere dense in $R(\Gamma)$ are nowhere dense in $R(\Gamma)$.*

(5) *Finite subsets of $R(\Gamma)$ are Γ-nowhere dense.*

Proof. Assertions (1), (2), (3a), and (3b) are obvious.

To prove (4a), let Z, Z' be Γ-nowhere dense sets, with h, $h' \in \Gamma_c$ as witnesses. Then $h \circ h' \in \Gamma_c$ testifies that $Z \cup Z'$ is Γ-nowhere dense. The result for arbitrary finite unions follows inductively. The proof of (4b) is identical.

To prove (5), fix $\gamma \in \Gamma_{cc}$. By (4a) it suffices to prove that the singleton set $\{R(\gamma)\}$ is Γ-nowhere dense. Define

$$h(x) = \begin{cases} x0 & \text{if } x1 \sqsubseteq \gamma(\gamma^{-1}(x)) \\ x1 & \text{otherwise.} \end{cases}$$

Then $h \in \Gamma_c$, so it suffices to show that $B_{h(x)} \cap \{R(\gamma)\} = \varnothing$ holds for all x. If $x \not\sqsubseteq R(\gamma)$ this is trivial, so fix $x \sqsubseteq R(\gamma)$. Let $z = \gamma(\gamma^{-1}(x))$. Then $z \sqsubseteq R(\gamma)$ and $z \not\sqsubseteq x \sqsubseteq R(\gamma)$, so $x \sqsubsetneq z$, so either $x0 \sqsubseteq z \sqsubseteq R(\gamma)$ or $x1 \sqsubseteq z \sqsubseteq R(\gamma)$. In the first case $h(x) = x1 \not\sqsubseteq R(\gamma)$; in the second case $h(x) = x0 \not\sqsubseteq R(\gamma)$. Either way, $B_{h(x)} \cap \{R(\gamma)\} = \varnothing$. □

We are finally ready to define what it means for a set to be "small" with respect to a notion of category Γ.

DEFINITION 3.9. A set $X \subseteq \mathcal{P}(\{0,1\}^*)$ is Γ-meager (respectively, *meager in $R(\Gamma)$*) if there exist a function $h \in \Gamma$ and a family $\{Z_k \mid k \in \mathbf{N}\}$ of sets such that

(i) $X \subseteq \cup\{Z_k \mid k \in \mathbf{N}\}$,

(ii) for each $k \in \mathbf{N}$, the function h_k testifies that Z_k is Γ-nowhere dense (respectively, nowhere dense in $R(\Gamma)$).

(Note that this implies $h \in \Gamma_{cc}$.) A set $X \subseteq \mathcal{P}(\{0,1\}^*)$ is *Γ-comeager* (respectively, *comeager in $R(\Gamma)$*) if its complement $\mathcal{P}(\{0,1\}^*) \setminus X$ is Γ-meager (respectively, meager in $R(\Gamma)$).

Thus a set X is Γ-meager if Γ provides sufficient resources to "uniformly enumerate" a family of Γ-nowhere dense sets which covers X. More concisely, X is Γ-meager if it is contained in a "Γ-union of Γ-nowhere dense sets."

If $\Gamma = \text{all}$, then "Γ-meager" and "meager in $R(\Gamma)$" are both equivalent to the classical notion of "meager set," i.e., "set of first category." If $\Gamma = \text{rec}$, then "meager in $R(\Gamma)$" is equivalent to the effective notion of meagerness of Mehlhorn [1973] and Lisagor [1979].

The rest of this section will develop the interpretation of meager sets as "small" subsets of $R(\Gamma)$. For this, we need the following definition.

DEFINITION 3.10. A *Γ-union* of Γ-meager sets (respectively, of sets meager in $R(\Gamma)$) is a set X such that there exist a function $g \in \Gamma$ and a family $\{X_k \mid k \in \mathbf{N}\}$ of sets such that

(i) $X = \cup\{X_k \mid k \in \mathbf{N}\}$,

(ii) for each $k \in \mathbf{N}$, the function g_k testifies that X_k is Γ-meager (respectively, meager in $R(\Gamma)$).

LEMMA 3.11. (0) *Γ-nowhere dense sets are Γ-meager and sets nowhere dense in $R(\Gamma)$ are meager in $R(\Gamma)$.*

(1–5) *Lemma* 3.8 *holds with "nowhere dense" replaced by "meager" throughout.*

(6a) *Γ-unions of Γ-meager sets are Γ-meager.*

(6b) *Γ-unions of sets meager in $R(\Gamma)$ are meager in $R(\Gamma)$.*

Proof. Assertions (0), (1), (2), (3), and (5) are obvious and assertion (4) follows trivially from assertion (6).

To prove (6a), assume X is a Γ-union of Γ-meager sets with $g \in \Gamma$ and $\{X_k \mid k \in \mathbf{N}\}$ as witnesses. Then for each k, g_k testifies that X_k is Γ-meager, i.e., there is a family $\{Z_{kj} \mid j \in \mathbf{N}\}$ of sets such that $X_k \subseteq \bigcup\{Z_{kj} \mid j \in \mathbf{N}\}$ and each $g_{kj} = (g_k)_j$ testifies that Z_{kj} is Γ-nowhere dense. Define $h(\langle\langle y, z\rangle, x\rangle) = g(\langle y, \langle z, x\rangle\rangle)$ and $Z'_n = Z_{\pi_1(n)\pi_2(n)}$. Then $h \in \Gamma$, $X = \bigcup\{X_k \mid k \in \mathbf{N}\} \subseteq \bigcup\{Z_{kj} \mid j, k \in \mathbf{N}\} = \bigcup\{Z'_n \mid n \in \mathbf{N}\}$, and each $h_n = g_{\pi_1(n)\pi_2(n)}$ testifies that Z'_n is Γ-nowhere dense, so X is Γ-meager.

The proof of (6b) is similar. □

Assertions (1) and (2) of Lemma 3.11, though obvious, are important because they unify results for various Γ. For example, if we prove that a set X is p-meager, then we know that it is meager in E, but we also know that it is pspace-meager, rec-meager, and all-meager; hence it is immediately meager in ESPACE, REC, and $\mathscr{P}(\{0, 1\}^*)$ as well. Thus a proof that X is p-meager yields considerably more information than a proof that it is meager in E. This is why results are usually stated in terms of Γ-meagerness, even when the matter of primary interest is meagerness in $R(\Gamma)$.

In the classical case, i.e., when $\Gamma = \text{all}$, a Γ-union is simply a countable union, so assertions (3) and (6) of Lemma 3.11 say that the meager sets form a *σ-ideal* of subsets of $\mathscr{P}(\{0, 1\}^*)$. Accordingly, in the general case, we interpret (3) and (6) as saying that the Γ-meager sets form a "Γ-ideal in $\mathscr{P}(\{0, 1\}^*)$" and that the sets meager in $R(\Gamma)$ form a "Γ-ideal in $R(\Gamma)$." Assertion (5) then tells us that these Γ-ideals contain many sets. (Note, however, that a singleton set $\{L\}$ need not be Γ-meager if $L \notin R(\Gamma)$.)

It is natural to think of the sets in a Γ-ideal as small, provided that the Γ-ideal is *proper*, i.e., does not contain every set. The following generalization of the classical Baire category theorem establishes this for the meager sets and thereby completes our argument that meager sets can be thought of as small sets. The simple diagonalization proof is a natural extension of the classical one.

THEOREM 3.12. (1) *A Γ-meager set contains no basic set.*

(2) *A set meager in $R(\Gamma)$ contains no basic set in $R(\Gamma)$.*

In particular, $\mathscr{P}(\{0, 1\}^*)$ is not Γ-meager and $R(\Gamma)$ is not meager in $R(\Gamma)$.

Proof. Assertion (1) follows easily from assertion (2) via part (1) of Lemma 3.11. (Alternatively, assertion (1) follows from the classical Baire category theorem via part (2) of Lemma 3.11.)

To prove (2), let $X \subseteq \mathscr{P}(\{0, 1\}^*)$ be meager in $R(\Gamma)$ with $h \in \Gamma_{cc}$ and $\{Z_k \mid k \in \mathbf{N}\}$ as witnesses and let $B'_z = B_z \cap R(\Gamma)$ be a basic set in $R(\Gamma)$. Define

$$\gamma(\langle w, x\rangle) = \begin{cases} h(\langle \lambda, z\rangle) & \text{if } x = \lambda \\ h(\langle w, x\rangle) & \text{if } x \neq \lambda, \end{cases}$$

and define the sequence $x_0 = \lambda$, $x_{k+1} = \gamma_k(x_k)$. Note five things:

(i) $\gamma \in \Gamma_{cc}$, so $R(\gamma) \in R(\Gamma)$.

(ii) $R(\gamma) \in B'_{x_k}$ for each $k \in \mathbf{N}$.

(iii) $R(\gamma) \in B'_{x_1} = B'_{h(\lambda, z)} \subseteq B'_z$.

(iv) $B'_{x_1} \cap Z_0 = B'_{h_0(z)} \cap Z_0 = \varnothing$, so $R(\gamma) \notin Z_0$.

(v) For $k \geqq 1$, $x_k \neq \lambda$, so $B'_{x_{k+1}} \cap Z_k = B'_{h_k(x_k)} \cap Z_k = \varnothing$, so $R(\gamma) \notin Z_k$.

These things together imply that

$$R(\gamma) \in B'_z \setminus \bigcup \{Z_k \mid k \in \mathbf{N}\} \subseteq B'_z \setminus X,$$

whence X does not contain B'_z. □

Thus the sets that are meager in $R(\Gamma)$ form a proper Γ-ideal.

4. Resource-bounded Banach–Mazur games. It is usually awkward to exhibit explicitly a Γ-meager set as a Γ-union of Γ-nowhere dense sets. In this section we give an alternate characterization of Γ-meager sets (and sets meager in $R(\Gamma)$) which is often easier to use when proving that a set is Γ-meager. The characterization is in terms of certain two-person infinite games of perfect information, called *Banach–Mazur games*. In the present setting, the games will be *resource-bounded* in the sense that a player may be required to play according to a strategy that can be computed within the resources provided by Γ. Thus the "perfect information" may not always be available in a usable form.

Informally, a Banach–Mazur game is an infinite game in which two players construct a language L by taking turns extending an initial bitmap of L. There is a distinguished set X of languages such that player I wins a play of the game if $L \in X$; player II wins otherwise.

More formally, a *strategy* for a Banach–Mazur game is a constructor σ. A *play* of a Banach–Mazur game is an ordered pair (α, β) of strategies. The *result* of the play (α, β) is the language $R(\alpha, \beta) = R(\beta \circ \alpha)$. This result is the language constructed when player I uses strategy α and player II uses strategy β. If $X \subseteq \mathcal{P}(\{0,1\}^*)$ and $\Sigma_{\mathrm{I}}, \Sigma_{\mathrm{II}}$ are classes of functions, then $\mathrm{G}[X; \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{II}}]$ is the Banach–Mazur game in which X is the distinguished set, player I is required to use a strategy $\alpha \in \Sigma_{\mathrm{I}}$, and player II is required to use a strategy $\beta \in \Sigma_{\mathrm{II}}$. A *winning strategy* for player I in $\mathrm{G}[X; \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{II}}]$ is a strategy $\alpha \in \Sigma_{\mathrm{I}}$ such that $R(\alpha, \beta) \in X$ holds for every $\beta \in \Sigma_{\mathrm{II}}$. A *winning strategy* for player II in $\mathrm{G}[X; \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{II}}]$ is a strategy $\beta \in \Sigma_{\mathrm{II}}$ such that $R(\alpha, \beta) \notin X$ holds for every $\alpha \in \Sigma_{\mathrm{I}}$. A player *wins* $\mathrm{G}[X; \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{II}}]$ if he has a winning strategy in $\mathrm{G}[X; \Sigma_{\mathrm{I}}, \Sigma_{\mathrm{II}}]$.

For the remainder of this section, let U be the set of all finite sequences of nonempty binary strings and, for $u = (x_1, \cdots, x_n) \in U$, let $\|u\| = |x_1| + \cdots + |x_n|$. For any strategy β, we define the associated function $\beta[\ ]$ from U into $\{0,1\}^*$ by the recursion $\beta[\lambda] = \lambda$, $\beta[x_1, \cdots, x_{n+1}] = \beta(\beta[x_1, \cdots, x_n] x_{n+1})$. We then define the sets

$$Y_k[\beta] = \bigcup \{B_{\beta[u]} \mid u \in U, \|u\| \geqq k\},$$

$$Y[\beta] = \bigcap \{Y_k[\beta] \mid k \in \mathbf{N}\},$$

$$Z_k[\beta] = \mathcal{P}(\{0,1\}^*) \setminus Y_k[\beta],$$

$$Z[\beta] = \mathcal{P}(\{0,1\}^*) \setminus Y[\beta] = \bigcup \{Z_k[\beta] \mid k \in \mathbf{N}\}.$$

Intuitively, $\beta[x_1, \cdots, x_n]$ is the status of a Banach–Mazur game immediately after player II's nth move if player I appends x_i to the bitmap in his ith move and player II uses strategy β. Thus $B_{\beta[x_1, \cdots, x_n]}$ is the set of all languages that could "conceivably" result from this play of the game, no matter what strategy either player uses after player II's nth move. In this same sense, $Y_k[\beta]$ is the set of all languages that could conceivably result from any play of the game in which player II uses strategy β in all moves up to and including his response to the move by which player I's total contribution to the bitmap reaches or exceeds k bits. Hence $Y[\beta]$ is the set of all languages that could result from any play of the game in which player II uses strategy β.

Note that $\beta[\ ]$ is recursive in β, but that $\beta[\ ]$ need not be computable within the resources of Γ, even when $\beta \in \Gamma$. (For example, if $|\beta(x)| = |x|^2$ for all x, then $|\beta[x_1, \cdots, x_n]|$ is hyperexponential in n.) Also note that $|\beta[u]| \geqq \|u\|$ and $\beta[u] \sqsubseteq \beta[v]$ hold for all $u, v \in U$ with u an initial subsequence of v.

LEMMA 4.1. *For any strategy $\beta \in \Gamma$, $Z[\beta]$ is Γ-meager.*

Proof. Let $\beta \in \Gamma_c$ and define

$$h(\langle x, y\rangle) = \begin{cases} \beta(x) & \text{if } y = \lambda \\ \beta(\beta(y)x) & \text{if } y \neq \lambda. \end{cases}$$

Then $h \in \Gamma$ and for each $k \in \mathbf{N}$ we have

$$\lambda \sqsubseteq 0^k \underset{\neq}{\sqsubseteq} \beta(0^k) = h(\langle 0^k, \lambda\rangle) = h_k(\lambda)$$

and

$$x \neq \lambda \Rightarrow x \underset{\neq}{\sqsubseteq} \beta(x) \sqsubseteq \beta(x)0^k \underset{\neq}{\sqsubseteq} \beta(\beta(x)0^k) = h(\langle 0^k, x\rangle) = h_k(x),$$

so $h \in \Gamma_{cc}$. Thus it suffices to show that each h_k testifies that $Z_k[\beta]$ is Γ-nowhere dense, i.e., that $B_{h_k(x)} \subseteq Y_k[\beta]$ holds for each k and x. This is trivial for $k = 0$ because $Y_0[\beta] = \mathcal{P}(\{0, 1\}^*)$, so assume $k > 0$. Then $B_{h_k(\lambda)} = B_{\beta(0^k)} = B_{\beta[0^k]} \subseteq Y_k[\beta]$ and for $x \neq \lambda$, $B_{h_k(x)} = B_{\beta(\beta(x)0^k)} = B_{\beta[x,0^k]} \subseteq Y_k[\beta]$, so we are finished in any case. □

If Γ is a notion of category, then it is easy to check that the class $\mathrm{rec}(\Gamma)$ of all functions f such that f is recursive in some $g \in \Gamma$ is also a notion of category.

LEMMA 4.2. (1) *If β is a strategy and $x \sqsubseteq L \in Y[\beta]$, then there exist $u \in U$ and $y \in \{0, 1\}^+$ such that $x \underset{\neq}{\sqsubseteq} \beta[u]y \underset{\neq}{\sqsubseteq} \beta(\beta[u]y) \sqsubseteq L$.*

(2) *If β is a strategy and $L \in Y[\beta]$, then there is a strategy α such that $R(\alpha, \beta) = L$.*

(3) *If $\beta \in \Gamma$ is a strategy and $L \in Y[\beta] \cap R(\Gamma)$, then there is a strategy $\alpha \in \mathrm{rec}(\Gamma)$ such that $R(\alpha, \beta) = L$.*

Proof. Assume the hypothesis of assertion (1). Then $L \in Y_{|x|+1}[\beta]$, so there exists $u' = (x_1, \cdots, x_n) \in U$ with $\|u'\| > |x|$ and $L \in B_{\beta[u']}$, i.e., $\beta[u'] \sqsubseteq L$. Note that $n \geqq 1$, let $u = (x_1, \cdots, x_{n-1})$, and let $y = x_n$. Then $|\beta[u]y| \geqq \|u'\| > |x|$ and $\beta[u]y \underset{\neq}{\sqsubseteq} \beta(\beta[u]y) = \beta[u'] \sqsubseteq L$. Since $x \sqsubseteq L$, it follows that $x \underset{\neq}{\sqsubseteq} \beta[u]y$; hence (1) holds.

Assertion (2) follows trivially from assertion (3) by taking $\Gamma = \text{all}$.

To prove (3), assume the hypothesis. Since $\beta \in \Gamma$ and $L \in R(\Gamma)$, the condition $x \underset{\neq}{\sqsubseteq} \beta[u]y \underset{\neq}{\sqsubseteq} \beta(\beta[u]y) \sqsubseteq L$ is decidable relative to Γ. It follows that there is a partial function g with the following properties.

(i) If there exist $u \in U$ and $y \in \{0, 1\}^+$ such that $x \underset{\neq}{\sqsubseteq} \beta[u]y \underset{\neq}{\sqsubseteq} \beta(\beta[u]y) \sqsubseteq L$, then $g(x)$ is defined and has the form $g(x) = \beta[u]y$ for some such u and y.

(ii) If no such u and y exist, then $g(x)$ is undefined.

(iii) g is partial recursive in Γ. (Briefly, g is computed by performing an unbounded search for such u and y.)

Define

$$\alpha(x) = \begin{cases} g(x) & \text{if } x \sqsubseteq L \\ x0 & \text{if } x \not\sqsubseteq L. \end{cases}$$

Since $L \in Y[\beta]$, (i) and part (1) of this lemma tell us that $\alpha(x)$ is defined for all x. Since $L \in R(\Gamma)$, the condition $x \sqsubseteq L$ is decidable relative to Γ. It follows by (iii) that $\alpha \in \mathrm{rec}(\Gamma)$. It is clear that $x \underset{\neq}{\sqsubseteq} \alpha(x)$ for all x, i.e., that α is a strategy. A routine induction shows that $R(\alpha, \beta) = L$. □

We now characterize the Γ-meager sets in terms of Banach–Mazur games.

THEOREM 4.3. *For a set* $X \subseteq \mathcal{P}(\{0,1\}^*)$, *consider the following conditions*:

(a) *Player* II *wins* $G[X; \text{all}, \Gamma]$.

(b) X *is* Γ*-meager.*

In any case, (a) *implies* (b). *If* Γ *has the Mazur property, then* (b) *implies* (a).

Proof. Assume (a) holds with the winning strategy $\beta \in \Gamma$ as witness. Assume $L \in Y[\beta]$. By part (2) of Lemma 4.2, there is a strategy α such that $R(\alpha, \beta) = L$. Since β is a winning strategy for player II, it follows that $L \notin X$. Taking the contrapositive, this argument shows that $X \subseteq Z[\beta]$. Since $Z[\beta]$ is Γ-meager by Lemma 4.1, it follows that (b) holds.

Conversely, assume that Γ has the Mazur property and that (b) holds, i.e., that X is Γ-meager with $h \in \Gamma_{cc}$ and $\{Z_k \mid k \in \mathbf{N}\}$ as witnesses. Note that if $|h^{\mathrm{II}}(x)| > k$, then there is a sequence $x_0, \cdots, x_k$ with

$$\lambda \underset{\neq}{\sqsubseteq} x_0 \underset{\neq}{\sqsubseteq} h_0(x_0) \underset{\neq}{\sqsubseteq} \cdots \underset{\neq}{\sqsubseteq} x_k \underset{\neq}{\sqsubseteq} h_k(x_k) \underset{\neq}{\sqsubseteq} x,$$

so $B_x \cap Z_k \subseteq B_{h_k(x_k)} \cap Z_k = \varnothing$, the last equality holding because h_k testifies that Z_k is Γ-nowhere dense. This shows that h^{II} has the property that

$$(*) \qquad |h^{\mathrm{II}}(x)| > k \Rightarrow B_x \cap Z_k = \varnothing$$

holds for all x and k. Define $\beta(x) = h(\langle h^{\mathrm{II}}(x), x\rangle)$. Since $h \in \Gamma_{cc}$ and Γ has the Mazur property, β is a strategy in Γ. To see that β wins $G[X; \text{all}, \Gamma]$ for player II, let α be an arbitrary strategy for player I. It is immediate from the definitions of h^{II} and β that $|h^{\mathrm{II}}((\beta \circ \alpha)^n(\lambda))|$ is strictly increasing in n. It follows by (*) that for each k there exists n such that $B_{(\beta\circ\alpha)^n(\lambda)} \cap Z_k = \varnothing$. Since $(\beta \circ \alpha)^n(\lambda) \sqsubseteq R(\alpha, \beta)$ for each n, it follows that $R(\alpha, \beta) \notin Z_k$ for each k. But then $R(\alpha, \beta) \notin X$, i.e., player II wins, so (a) holds. □

Analogously, we characterize the sets which are meager in $R(\Gamma)$.

THEOREM 4.4. *For a set* $X \subseteq \mathcal{P}(\{0,1\}^*)$, *consider the following conditions*:

(a) *Player* II *wins* $G[X \cap R(\Gamma); \text{all}, \Gamma]$.

(b) *Player* II *wins* $G[X \cap R(\Gamma); \text{rec}(\Gamma), \Gamma]$.

(c) X *is meager in* $R(\Gamma)$.

In any case, (a) *implies* (b) *and* (b) *implies* (c). *If* Γ *has the Mazur property, then* (c) *implies* (a).

Proof. It is trivial that (a) implies (b).

The proof that (b) implies (c) is the same as the proof that (a) implies (b) in Theorem 4.3, except that $X \cap R(\Gamma)$ is used in place of X and part (3) of Lemma 4.2 is used in place of part (2).

If Γ has the Mazur property, then the proof that (c) implies (a) is the same as the proof that (b) implies (a) in Theorem 4.3, except that $X \cap R(\Gamma)$ is used in place of X and basic sets $B_z \cap R(\Gamma)$ in $R(\Gamma)$ are used in place of basic sets B_z. □

In the case where Γ has the Mazur property, the equivalence of (a) and (b) in Theorem 4.4 is somewhat remarkable. For example, if $\Gamma = \text{pspace}$ and $X \subseteq \text{ESPACE}$, this says that player II wins $G[X; \text{all}, \text{pspace}]$ if he wins $G[X; \text{rec}, \text{pspace}]$. That is, if player II can beat every recursive strategy, he can beat any strategy whatsoever. Intuitively, this says that, in the game $G[X; \text{all}, \text{pspace}]$, most of player I's available resources are no help to him.

The role of the Mazur property in Theorems 4.3 and 4.4 illustrates an interesting aspect of the resource-bounded setting. In the classical and effective settings (where the Mazur property holds trivially), the fact that player II can win whenever the designated set is meager is the "easy direction" of the characterization. (This direction was noted by Mazur when he invented the classical game; it was Banach who subsequently proved the converse.) In the resource-bounded setting, this direction seems

to require an additional property, which we have called the Mazur property. For example, it is not clear that player II wins G[X; rec, p], or even G[X; p, p], whenever X is a meager subset of E.

We conclude this section with an easy application. A language L is *sparse* if there is a polynomial q such that $|L_{=n}| \leqq q(n)$ for all n. The sparse languages are a central concern of current research in computational complexity theory. It is easy to see that the set SPARSE of all sparse languages has the cardinality of the continuum, i.e., of $\mathscr{P}(\{0,1\}^*)$. The following theorem shows that SPARSE is nevertheless small in the polynomial time-bounded sense of category.

THEOREM 4.5. SPARSE *is* p-*meager*; *hence, it is meager in* E.

Proof. Consider the strategy β that extends x by appending $4|x|+1$ 1's to it. It is clear that β is a constructor and $\beta \in$ p. It is also easy to check that, for any strategy α, there are infinitely many n such that $|R(\alpha, \beta)_{=n}| = 2^n$, whence $R(\alpha, \beta) \notin$ SPARSE. Thus β wins G[SPARSE; all, p] for player II, so the present theorem follows from Theorem 4.3. □

5. Resource-bounded measure. The sense in which meager sets are small is not always the most intuitive one. For example, consider the set X of all languages L such that $xx \sqsubseteq L$ holds only for finitely many strings x. The strategy $\beta(x) = xx$ testifies readily that X is p-meager. However, if L is chosen probabilistically by using an independent toss of a fair coin to decide whether each $s_i \in L$, then it is easy to see that L will be in X with probability 1, i.e., almost certainly. Thus the Baire category notion of smallness disagrees sharply with this very intuitive probability measure on $\mathscr{P}(\{0,1\}^*)$.

This independent coin-toss measure is precisely the classical Lebesgue measure on $\mathscr{P}(\{0,1\}^*)$. That is, if we identify a language L with the real number $x \in [0,1]$, whose binary expansion is the characteristic bitstring of L, then the measure of a set of languages is (when defined) precisely the usual Lebesgue measure of the corresponding subset of the unit interval. Equivalently, this measure is the product probability measure on $\prod_{x \in \{0,1\}^*} \{0,1\}$, where $\{0,1\}$ has the uniform distribution.

In this section we introduce and develop *resource-bounded* (*Lebesgue*) *measure*, i.e., *resource-bounded probability*, for complexity classes of languages. This will provide a notion of smallness for subsets of these classes that corresponds nicely with the classical Lebesgue measure on $\mathscr{P}(\{0,1\}^*)$.

The subject of this paper is resource-bounded category and measure in exponential complexity classes. It will thus suffice here to say that a *notion of measure* is a class Δ of functions from $\{0,1\}^*$ into $\{0,1\}^*$ and that the classes all, rec, $\mathrm{p}_i (i \geqq 1)$, and $\mathrm{p}_i$space $(i \geqq 1)$ are notions of measure. Some results will be stated in terms of arbitrary notions of measure Δ, but we will require their proofs to be valid for these examples only. This approach is less general than that of § 3 but is still general enough to encompass classical, effective, and resource-bounded notions.

From now on, Δ, Δ', etc., will denote notions of measure in the above sense. The result classes $R(\Delta)$ are defined exactly as in § 3, so the language classes treated here are $\mathscr{P}(\{0,1\}^*)$, REC, $\mathrm{E}_i (i \geqq 1)$, and E_iSPACE $(i \geqq 1)$.

DEFINITION 5.1. The *measure* of a basic set B_x is $\mu(B_x) = 2^{-|x|}$. The *measure* of the empty set is $\mu(\varnothing) = 0$.

DEFINITION 5.2. A Δ-*cover* is a pair $(h, m) \in \Delta^2$ such that $m(\{0\}^*) \subseteq \{0\}^*$ and

$$\sum_{k=|m(0^l)|}^{\infty} \mu(B_{h(0^k)}) \leqq 2^{-l}$$

holds for each $l \in \mathbf{N}$. If (h, m) is a Δ-cover, then h is called the *enumerator*, m is called

the *modulus*, and the real number

$$\mu^*(h) = \sum_{k=0}^{\infty} \mu(B_{h(0^k)})$$

exists and is called the *total measure* of (h, m).

Intuitively, a Δ-cover is a family of basic sets $B_{h(\lambda)}, B_{h(0)}, B_{h(00)}, \cdots$ such that Δ provides sufficient resources to enumerate the family and to compute approximations of the finite total measure of the family.

DEFINITION 5.3. A Δ-*cover* of a set $X \subseteq \mathcal{P}(\{0,1\}^*)$ is a Δ-cover (h, m) such that $X \subseteq \cup\{B_{h(0^k)} \mid k \in \mathbf{N}\}$.

DEFINITION 5.4. A Δ-*null cover* of a set $X \subseteq \mathcal{P}(\{0,1\}^*)$ is a pair $(h, m) \in \Delta^2$ such that the following two conditions hold for each $k \in \mathbf{N}$:

(i) (h_k, m_k) is a Δ-cover of X.

(ii) $\mu^*(h_k) \leqq 2^{-k}$.

DEFINITION 5.5. Let $X \subseteq \mathcal{P}(\{0,1\}^*)$ and let $X^c = \mathcal{P}(\{0,1\}^*) \setminus X$ be the complement of X.

(1) X has Δ-*measure* 0, and we write $\mu_\Delta(X) = 0$, if there exists a Δ-null cover of X.

(2) X has *measure* 0 *in* $R(\Delta)$, and we write $\mu(X \mid R(\Delta)) = 0$, if $\mu_\Delta(X \cap R(\Delta)) = 0$.

(3) X has Δ-*measure* 1, and we write $\mu_\Delta(X) = 1$, if $\mu_\Delta(X^c) = 0$.

(4) X has *measure* 1 in $R(\Delta)$, and we write $\mu(X \mid R(\Delta)) = 1$, if $\mu(X^c \mid R(\Delta)) = 0$.

Thus a set X of languages has Δ-measure 0 if Δ contains sufficient resources to uniformly enumerate Δ-covers of X with rapidly vanishing total measure.

Note that $\mu(X \mid R(\Delta))$ depends only on the set $X \cap R(\Delta)$. In particular, the conditions $\mu(X \mid R(\Delta)) = 1$ and $\mu_\Delta(X \cap R(\Delta)) = 1$ are *not* equivalent.

It is amusing to think of $\mu(X \mid R(\Delta))$ as the "conditional probability" that $L \in X$, given that $L \in R(\Delta)$, when L is chosen by independent tosses of a fair coin. It should be emphasized, however, that this interpretation is not meaningful (and is probably misleading) because, in cases of interest, $R(\Delta)$ will be a countable, hence measure 0, subset of $\mathcal{P}(\{0,1\}^*)$.

The next definition and lemma are analogous to Definition 3.10 and Lemma 3.11.

DEFINITION 5.6. *A* Δ-*union* of Δ-measure 0 sets (respectively, of sets of measure 0 in $R(\Delta)$) is a set X such that there exist a pair $(h, m) \in \Delta^2$ and a family $\{X_k \mid k \in \mathbf{N}\}$ of sets such that

(i) $X = \cup\{X_k \mid k \in \mathbf{N}\}$,

(ii) for each $k \in \mathbf{N}$, the pair $(h_k, m_k) \in \Delta^2$ testifies that X_k has Δ-measure 0 (respectively, has measure 0 in $R(\Delta)$).

Of course any finite union is a Δ-union here.

LEMMA 5.7. (1) *If* X *has* Δ-*measure* 0, *then* X *has measure* 0 *in* $R(\Delta)$.

(2) *If* X *has* Δ-*measure* 0 *and* $\Delta \subseteq \Delta'$, *then* X *has* Δ'-*measure* 0.

(3a) *Subsets of* Δ-*measure* 0 *sets have* Δ-*measure* 0.

(3b) *Subsets of sets with measure* 0 *in* $R(\Delta)$ *have measure* 0 *in* $R(\Delta)$.

(4a) Δ-*unions of* Δ-*measure* 0 *sets have* Δ-*measure* 0.

(4b) Δ-*unions of sets with measure* 0 *in* $R(\Delta)$ *have measure* 0 *in* $R(\Delta)$.

(5) *Finite subsets of* $R(\Delta)$ *have* Δ-*measure* 0.

Proof. Assertions (1), (2), and (3) are clear and (4b) follows immediately from (4a), so it suffices to prove (4a) and (5).

To prove (4a), let $(h, m) \in \Delta^2$ and the family $\{X_k \mid k \in \mathbf{N}\}$ testify that X is a Δ-union of Δ-measure 0 sets. Define

$$h'(\langle u, \langle v, w\rangle\rangle) = h(\langle v, \langle uv0, w\rangle\rangle).$$

Then $h' \in \Delta$ and for each $k, i, j \in \mathbf{N}$, $h'_k(\langle 0^i, 0^j\rangle) = h_{i,k+i+1}(0^j)$. That is, h'_k "weaves together" the enumerators $h_{i,k+i+1}$ for $i \in \mathbf{N}$. Note that each $(h_{i,k+i+1}, m_{i,k+i+1})$ is a Δ-cover of X_i with total measure $\mu^*(h_{i,k+i+1}) \leqq 2^{-(k+i+1)}$. Define m' so that $m'(\{0,1\}^*) \subseteq \{0\}^*$ and

$$|m'(\langle 0^k, 0^l\rangle)| = 1 + \max\{\langle i, j\rangle \mid i \leqq l \text{ and } j \leqq |m_{i,k+i+1}(0^{2l+1})|\}$$

for all $k, l \in \mathbf{N}$. The key property of m' here is that

(i) for each $i, j, k, l \in \mathbf{N}$, the condition $\langle i, j\rangle \geqq |m'_k(0^l)|$ implies that $i \geqq l+1$ or $j \geqq |m_{i,k+i+1}(0^{2l+1})|$.

It is clear that $(h', m') \in \Delta^2$. We will show that (h', m') is in fact a Δ-null cover of X. For this, it suffices to prove that the following three things hold for each $k, l \in \mathbf{N}$:

(a) $X \subseteq \bigcup_{n\in\mathbf{N}} B_{h'_k}(0^n)$.

(b) $\sum_{n=|m'_k(0^l)|}^{\infty} \mu(B_{h'_k(0^n)}) \leqq 2^{-l}$.

(c) $\mu^*(h'_k) \leqq 2^{-k}$.

So fix $k, l \in \mathbf{N}$. To prove (a), just note that each $X_i \subseteq \bigcup_{j\in\mathbf{N}} B_{h_{i,k+i+1}(0^j)} = \bigcup_{j\in\mathbf{N}} B_{h'_k(\langle 0^i,0^j\rangle)}$, whence $X = \bigcup_{i\in\mathbf{N}} X_i \subseteq \bigcup_{i,j\in\mathbf{N}} B_{h'_k(\langle 0^i,0^j\rangle)} = \bigcup_{n\in\mathbf{N}} B_{h'_k(0^n)}$. For convenience, write $\hat{m} = |m_{i,k+i+1}(0^{2l+1})|$. Before proving (b) and (c), note that the following two things hold for each $i \in \mathbf{N}$.

(ii) $\sum_{j=0}^{\infty} \mu(B_{h'_k(\langle 0^i,0^j\rangle)}) = \mu^*(h_{i,k+i+1})$.

(iii) $\sum_{j=\hat{m}}^{\infty} \mu(B_{h'_k(\langle 0^i,0^j\rangle)}) \leqq 2^{-(2l+1)}$.

Now to prove (b), note that (i) tells us that

$$\sum_{n=|m'_k(0^l)|}^{\infty} \mu(B_{h'_k(0^n)}) \leqq \sum_{i=l+1}^{\infty}\sum_{j=0}^{\infty} \mu(B_{h'_k(\langle 0^i,0^j\rangle)}) + \sum_{i=0}^{l}\sum_{j=\hat{m}}^{\infty} \mu(B_{h'_k(\langle 0^i,0^j\rangle)}),$$

whence by (ii) and (iii) we have

$$\begin{aligned}\sum_{n=|m'_k(0^l)|}^{\infty} \mu(B_{h'_k(0^n)}) &\leqq \sum_{i=l+1}^{\infty} \mu^*(h_{i,k+i+1}) + \sum_{i=0}^{l} 2^{-(2l+1)} \\ &\leqq \sum_{i=l+1}^{\infty} 2^{-(k+i+1)} + (l+1)2^{-(2l+1)} \leqq 2^{-l},\end{aligned}$$

i.e., (b) holds. Finally, (ii) tells us that

$$\mu^*(h'_k) = \sum_{i=0}^{\infty}\sum_{j=0}^{\infty} \mu(B_{h'_k(\langle 0^i,0^j\rangle)}) = \sum_{i=0}^{\infty} \mu^*(h_{i,k+i+1}) \leqq \sum_{i=0}^{\infty} 2^{-(k+i+1)} = 2^{-k},$$

i.e., (c) holds. Thus (h', m') is a Δ-null cover of X, so (4a) is affirmed.

To prove (5), it is sufficient by (4a) to prove that singleton subsets of $R(\Delta)$ have Δ-measure 0. So let $L = R(\delta)$, where $\delta \in \Delta$. Define $h(\langle u, v\rangle) = \delta^n(\lambda)$, where n is least such that $|\delta^n(\lambda)| > |uv|$, and $m(\langle u, v\rangle) = v$. Then it is easy to check that (h, m) is a Δ-null cover of $\{L\}$. □

Assertions (1) and (2) of Lemma 5.7 have the same unifying effect as the corresponding assertions of Lemma 3.11. Thus, for example, if we prove that a set $X \subseteq \mathcal{P}(\{0,1\}^*)$ has p-measure 0, then we can conclude immediately that $0 = \mu(X \mid \mathrm{E}) = \mu(X \mid \mathrm{ESPACE}) = \mu(X \mid \mathrm{REC}) = \mu(X)$.

Assertions (3) and (4) of Lemma 5.7 say that the Δ-measure 0 sets form a Δ-ideal in $\mathcal{P}(\{0,1\}^*)$ and that the sets that have measure 0 in $R(\Delta)$ form a Δ-ideal in $R(\Delta)$. Assertion (5) then says that these Δ-ideals contain many sets.

The following theorem is analogous to Theorem 3.12 in that it shows that these Δ-ideals are proper. The main idea of the proof is a diagonalization, which we first isolate in a lemma. (This lemma is a resource-bounded version of a classical theorem of Borel.)

LEMMA 5.8. *If $X \subseteq R(\Delta)$ has a Δ-cover with total measure less than $2^{-|z|}$, then X does not contain $B_z \cap R(\Delta)$.*

Proof. Let (h, m) be a Δ-cover of $X \subseteq R(\Delta)$ with $\mu^*(h) < 2^{-|z|}$ and let $X' = \cup\{B_{h(0^k)} \mid k \in \mathbf{N}\}$. Note that $X \subseteq X'$. Fix a rational q and a positive integer l such that $2^{|z|}\mu^*(h) \leqq q < q + 2^{-l} < 1$. Define from h a real-valued "density function"

$$d(x) = 2^{|x|} \sum_{k=0}^{\infty} \mu(B_x \cap B_{h(0^k)})$$

and an "approximate density function"

$$e(x, n) = 2^{|x|} \sum_{k=0}^{n-1} \mu(B_x \cap B_{h(0^k)}).$$

Note that each $e(x, n) \leqq d(x)$ and $\lim_{n\to\infty} e(x, n) = d(x)$. Note also that for each $x \in \{0, 1\}^*$,

(i) $B_x \subseteq X'$ implies $d(x) \geqq 1$, and

(ii) $d(x) = \frac{1}{2}[d(x0) + d(x1)]$.

Define a clocked constructor δ by

$$\delta_k(x) = \begin{cases} x0 & \text{if } e(x0, \hat{m}) \leqq q + 2^{-l}(1 - 2^{-k}) \\ x1 & \text{otherwise,} \end{cases}$$

where $\hat{m} = |m(0^{|x|+k+l+2})|$. Since $h, m \in \Delta$, it is easy to check that $\delta \in \Delta$.

Assume for a moment that $d(x) \leqq q + 2^{-l}(1 - 2^{-k})$. Then (ii) tells us that $\min\{d(x0), d(x1)\} \leqq q + 2^{-l}(1 - 2^{-k})$, whence $e(\delta_k(x), \hat{m}) \leqq q + 2^{-l}(1 - 2^{-k})$. It follows that

$$\begin{aligned} d(\delta_k(x)) &\leqq e(\delta_k(x), \hat{m}) + 2^{|x|+1} \sum_{j=\hat{m}}^{\infty} 2^{-|h(0^j)|} \\ &\leqq q + 2^{-l}(1 - 2^{-k}) + 2^{|x|+1} 2^{-|x|-k-l-2} \\ &= q + 2^{-l}(1 - 2^{-k-1}). \end{aligned}$$

Now define the clocked constructor $\hat{\delta}$ by $\hat{\delta}_k(x) = \delta_k(\hat{x})$, where $\hat{x} = z$ if $x \sqsubseteq z$ and $\hat{x} = x$ otherwise, and define the sequence $x_0 = \lambda$, $x_{k+1} = \hat{\delta}_k(x_k)$. It is clear that $\hat{\delta} \in \Delta$ and that $z \sqsubseteq x_1$, whence $R(\hat{\delta}) \in B_z$. It is also clear that $d(z) \leqq 2^{|z|}\mu^*(h) \leqq q$, whence the preceding paragraph provides an inductive proof that

(iii) $d(x_k) \leqq q + 2^{-l}(1 - 2^{-k})$

holds for each $k \geqq 1$.

Now let $k \in \mathbf{N}$ be arbitrary and let $\hat{k} = |h(0^k)| + 1$. Then (iii) tells us that $d(x_{\hat{k}}) \leqq q + 2^{-l} < 1$, so $B_{x_{\hat{k}}} \not\subseteq X'$ by (i). In particular, then, $B_{x_{\hat{k}}} \not\subseteq B_{h(0^k)}$. Since $|x_{\hat{k}}| \geqq \hat{k} > |h(0^k)|$ and $R(\hat{\delta}) \in B_{x_{\hat{k}}}$, it follows that $R(\hat{\delta}) \notin B_{h(0^k)}$. Since k is arbitrary here, it follows that $R(\hat{\delta}) \notin X'$, whence $R(\hat{\delta}) \notin X$.

We now have $\hat{\delta} \in \Delta$ such that $R(\hat{\delta}) \in B_z \setminus X$. It follows that X does not contain $B_z \cap R(\Delta)$. □

THEOREM 5.9.

(1) *A Δ-measure 0 set contains no basic set.*

(2) *A set of measure 0 in $R(\Delta)$ contains no basic set in $R(\Delta)$.*

Proof. Assertion (1) is immediate from assertion (2) via part (1) of Lemma 5.7.

To prove (2), let the null cover (h, m) testify that $\mu(X \mid R(\Delta)) = 0$ and fix a basic set $B_z \cap R(\Delta)$ in $R(\Delta)$. Then the pair $(h_{|z|+1}, m_{|z|+1})$ is a Δ-cover of $X \cap R(\Delta)$ with total measure $\mu^*(h_{|z|+1}) \leqq 2^{-|z|-1} < 2^{-|z|}$. It follows by Lemma 5.8 that X does not contain $B_z \cap R(\Delta)$. □

By Lemma 5.7 and Theorem 5.9, the measure 0 subsets of $R(\Delta)$ are "small" subsets of $R(\Delta)$.

Once again let X be the set considered in the first paragraph of this section. For each $k, l \in \mathbf{N}$, let $h_k(0^l) = xx$, where x is the lth binary string of length $> k$, and let $m_k(0^l) = 0^{2^{l+1}}$. It is easy to check that (h, m) is a pspace-null cover of the complement of X, whence X has pspace-measure 1. On the other hand, we saw that X is p-meager, hence certainly pspace-meager. It follows that X is meager and has measure 1 in ESPACE. Thus, just as in the classical case, resource-bounded notions of category and measure do not always agree as to which sets are small.

It is conspicuous that the resource-bounded measure theory developed so far assigns measures of only 0 or 1 to sets of languages. We now present the basic ideas of a more comprehensive resource-bounded measure theory and explain why sets of intermediate measures are of very little interest in complexity theory.

The following definition is a generalization of Lebesgue's original formulation of measurability via inner and outer measure.

DEFINITION 5.10. Let $X \subseteq \mathcal{P}(\{0,1\}^*)$ and let $X^c = \mathcal{P}(\{0,1\}^*) \setminus X$ be the complement of X.

(1) X is *Δ-measurable* if there is a triple $(g, h, m) \in \Delta^3$ such that the following conditions all hold for each $k \in \mathbf{N}$:

(i) (g_k, m_k) is a Δ-cover of X,

(ii) (h_k, m_k) is a Δ-cover of X^c,

(iii) $\mu^*(g_k) + \mu^*(h_k) \leqq 1 + 2^{-k}$.

In this case, the real number

$$\mu_\Delta(X) = \lim_{k\to\infty} \mu^*(g_k) = 1 - \lim_{k\to\infty} \mu^*(h_k)$$

exists and is called the *Δ-measure* of X.

(2) X is *measurable in* $R(\Delta)$, i.e., X is an *event in* $R(\Delta)$, if there is a triple $(g, h, m) \in \Delta^3$ such that conditions (i'), (ii'), and (iii) hold for each $k \in \mathbf{N}$, where (i') and (ii') are conditions (i) and (ii) above, with X and X^c replaced by $X \cap R(\Delta)$ and $R(\Delta) \setminus X$, respectively. In this case, the real number

$$\mu(X \mid R(\Delta)) = \lim_{k\to\infty} \mu^*(g_k) = 1 - \lim_{k\to\infty} \mu^*(h_k)$$

exists and is called the *measure* of X *in* $R(\Delta)$.

It is easy to see that if X is Δ-measurable (respectively, measurable in $R(\Delta)$), then $\mu_\Delta(X)$ (respectively, $\mu(X \mid R(\Delta))$) is well defined, i.e., does not depend on the witness (g, h, m). Also, each of the conditions $\mu_\Delta(X) = 0$, $\mu_\Delta(X) = 1$, $\mu(X \mid R(\Delta)) = 0$, $\mu(X \mid R(\Delta)) = 1$ holds under Definition 5.10 if and only if it holds under Definition 5.5.

If $\Delta = \text{all}$, then Δ-measurability and measurability in $R(\Delta)$ are equivalent to each other and to classical Lebesgue measurability in $\mathcal{P}(\{0,1\}^*)$. Similarly, a set is measurable in $R(\text{rec}) = \text{REC}$ if and only if it is effectively measurable in the sense of Freidzon [1972].

For an easy resource-bounded example, one can check that each B_x is p-measurable with $\mu_p(B_x) = \mu(B_x)$. It follows easily that each B_x is measurable in E with $\mu(B_x \mid \text{E}) = \mu(B_x)$. As a cautionary example, however, note that $B_x \cap \text{E}$ is not p-measurable for any x.

We now give two useful lemmas. The first is immediate from Definition 5.10.

LEMMA 5.11. (1) *If X is Δ-measurable, then X is measurable in $R(\Delta)$ and* $\mu(X \mid R(\Delta)) = \mu_\Delta(X)$.

(2) *If X is Δ-measurable and $\Delta \subseteq \Delta'$, then X is Δ'-measurable and* $\mu_{\Delta'}(X) = \mu_\Delta(X)$.

LEMMA 5.12. (1) *If X is Δ-measurable and (h, m) is a Δ-cover of X, then $\mu^*(h) \geqq \mu_\Delta(X)$.*

(2) *If X is measurable in $R(\Delta)$ and (h, m) is a Δ-cover of $X \cap R(\Delta)$, then $\mu^*(h) \geqq \mu(X \mid R(\Delta))$.*

Proof. Assertion (1) follows immediately from assertion (2) by part (1) of Lemma 5.11.

To prove (2), let (g', h', m') testify that X is measurable in $R(\Delta)$ and let $k \in \mathbf{N}$ be arbitrary. Then (h'_k, m'_k) is a Δ-cover of $R(\Delta) \setminus X$ with $\mu^*(h'_k) \leqq 2^{-k} + \lim_{n\to\infty} \mu^*(h'_n)$. Now (h, m) and (h'_k, m'_k) can be "woven together" to give a Δ-cover (h'', m'') of $R(\Delta)$ with $\mu^*(h'') = \mu^*(h) + \mu^*(h'_k)$. By Lemma 5.8, then, $1 \leqq \mu^*(h'') = \mu^*(h) + \mu^*(h'_k)$, so $\mu^*(h) \geqq 1 - \mu^*(h'_k) \geqq 1 - \lim_{n\to\infty} \mu^*(h'_n) - 2^{-k} = \mu(X \mid R(\Delta)) - 2^{-k}$. Since k is arbitrary here, it follows that $\mu^*(h) \geqq \mu(X \mid R(\Delta))$. □

The following definition formalizes what it means for a class of languages to be "insensitive to finite alterations."

DEFINITION 5.13. (1) Two languages $L_1, L_2 \subseteq \{0,1\}^*$ are *equivalent almost everywhere*, and we write $L_1 \equiv L_2$ almost everywhere, if their symmetric difference $L_1 \Delta L_2$ is finite.

(2) A set $X \subseteq \mathcal{P}(\{0,1\}^*)$ is a *tail set* if for all $L_1, L_2 \subseteq \{0,1\}^*$ such that $L_1 \equiv L_2$ almost everywhere, $L_1 \in X$ if and only if $L_2 \in X$.

In complexity theory, virtually all language classes of interest are tail sets. In classical measure theory, the Kolmogorov [1933] zero-one law states that every measurable tail set has measure 0 or 1. We now prove a resource-bounded generalization of this law. We first need a lemma.

LEMMA 5.14. *If X is a tail set that has a Δ-cover of total measure <1, then X has Δ-covers of arbitrarily small total measure.*

Proof. Let (h, m) be a Δ-cover of the tail set X with $\mu^*(h) = r < 1$. Fix k such that $r^2 + 2^{-k} < r^{3/2}$ and let n be the maximum of all $|h(0^i)|$ for $0 \leqq i < \hat{m}$, where $\hat{m} = |m(0^k)|$. Let $w_0, w_1, \cdots, w_{s-1}$ be a list of distinct strings of length n such that $\bigcup\{B_{w_i} \mid 0 \leqq i < s\} = \bigcup\{B_{h(0^i)} \mid 0 \leqq i < \hat{m}\}$. Define from h a real-valued density function d exactly as in the proof of Lemma 5.8. Note that $d(w_i) \geqq 1$ for $0 \leqq i < s$. Fix a string u of length n such that $d(u) \leqq r$. (This u exists because $d(\lambda) = r$ and each $d(x) = \frac{1}{2}[d(x0) + d(x1)]$.)

Now modify the list $h(\lambda), h(0), h(0^2), \cdots$ as follows:

(i) Delete the first $\hat{m}$ entries.

(ii) In each place where there is an entry of the form uv, insert the entries $w_0 v, \cdots, w_{s-1} v$ immediately after.

Since k, $\hat{m}$, n, s, the list $w_0, \cdots, w_{s-1}$, and u are all constants, there is an enumerator $h' \in \Delta$ such that the resulting list is exactly $h'(\lambda), h'(0), h'(0^2), \cdots$. Also, the function $m' \in \Delta$ defined by $m'(x) = m(x0^{s+1})^{s+1}$ is clearly a modulus for h', i.e., (h', m') is a Δ-cover. In fact, since X is a tail set, (h, m) is a Δ-cover of X, and h' replicates on each $X \cap B_{h(0^i)}$, $0 \leqq i < \hat{m}$, the cover of $X \cap B_u$ by h, (h', m') is a Δ-cover of X. This new Δ-cover of X has total measure

$$\mu^*(h') = s2^{-n} d(u) + \sum_{i=\hat{m}}^{\infty} \mu(B_{h(0^i)}) \leqq \mu^*(h) d(u) + 2^{-k} \leqq r^2 + 2^{-k} < r^{3/2}.$$

We have now shown that if $r < 1$ and X has a Δ-cover of total measure r, then it has a Δ-cover of total measure of $< r^{3/2}$. Since the sequence $r_0 = r$, $r_{n+1} = r_n^{3/2}$ converges to 0, this proves the lemma. □

We now prove the zero-one law for resource-bounded measure.

THEOREM 5.15. (1) *If X is a Δ-measurable tail set, then $\mu_\Delta(X) = 0$ or $\mu_\Delta(X) = 1$.*

(2) *If X is a tail set that is measurable in $R(\Delta)$, then $\mu(X \mid R(\Delta)) = 0$ or $\mu(X \mid R(\Delta)) = 1$.*

Proof. Assertion (1) follows immediately from assertion (2) by part (1) of Lemma 5.11.

To prove (2), let X be a tail set that is measurable in $R(\Delta)$ and assume that $\mu(X \mid R(\Delta)) < 1$. Let $\varepsilon > 0$ be arbitrary. By Definition 5.10, $X \cap R(\Delta)$ has a Δ-cover of total measure <1, whence by Lemma 5.14, it has a Δ-cover (h, m) with $\mu^*(h) < \varepsilon$. By part (2) of Lemma 5.12, then, $\mu(X \mid R(\Delta)) < \varepsilon$. Thus $\mu(X \mid R(\Delta)) = 0$. □

As we have noted, most sets of interest in complexity theory are tail sets. By Theorem 5.15, every measurable tail set in $R(\Delta)$ has measure 0 or 1 in $R(\Delta)$, so sets of intermediate measure are of very little complexity-theoretic interest. We should emphasize, however, that for resource-bounded notions Δ, measurability in $R(\Delta)$ is a very strong hypothesis. Thus we do not interpret Theorem 5.15 to mean that sets of interest necessarily have measure 0 or 1. The third possibility, nonmeasurability in $R(\Delta)$, must be considered.

In this section we have presented a mere beginning of resource-bounded measure theory. We have not selected an axiomatization, we have restricted attention to measures induced by the usual Lebesgue measure on $\mathcal{P}(\{0,1\}^*)$, and we have omitted even the most basic properties of measurable sets (e.g., they form a "Δ-algebra," the measure is monotone and "Δ-additive," etc.). The only theorems we have proven are 5.9, the nontriviality of the measure, and 5.15, the resource-bounded zero-one law. Nevertheless, we have enough to begin applying the theory to the structure of exponential complexity classes.

6. Resource-bounded Kolmogorov complexity. Theorem 1.2 says that some languages in ESPACE have high space-bounded Kolmogorov complexity. In this section we prove that, with respect to both category and measure, nearly all languages in ESPACE have this property. We then prove an analogous but weaker result for exponential time complexity classes.

THEOREM 6.1. *For any $c > 0$ and $b < 1$, the set of all languages L such that $KS[2^{cn}](L_{\leqq n}) < b \cdot 2^{n+1}$ almost everywhere is* pspace-*meager and has* pspace-*measure* 0.

Proof. Let X be the set of all such L, where we assume without loss of generality that c is a positive integer and b is a rational between 0 and 1.

To see that X is pspace-meager, it suffices by Theorem 4.3 to exhibit a winning strategy β for player II in the game G[X; all, pspace]. Let β be defined as follows. (Recall that $s_0, s_1, s_2, \cdots$ is the standard enumeration of $\{0,1\}^*$.)

begin
  input x;
  $n :=$ the least integer such that $(1-b)2^{n+1} \geqq |x|+1$;
  while $|x| < 2^{n+1} - 1$ **do**
  begin //decide $s_{|x|}$//
    VOTE;
    if yes $< \frac{1}{2}$(total)
      then $x := x1$
      else $x := x0$
  end while;
  output x
end β.

The macro VOTE operates as follows.

```
begin
    yes, total, π := 0, 0, λ;
    while |π| < b · 2^{n+1} do
    begin
        if OK (π, n, x) then
        begin // π gets to vote//
            total := total + 1;
            if U(⟨π, s_{|x|}⟩) outputs 1 in ≦2^{cn} space
                then yes := yes + 1
        end if;
        π := next (π)
    end while
end VOTE.
```

The predicate $OK(\pi, n, x)$ here asserts that $|x| < 2^{n+1} - 1$, $U(\langle \pi, s_i \rangle)$ halts in $\leqq 2^{cn}$ space for each $0 \leqq i \leqq |x|$, and $U(\langle \pi, s_i \rangle)$ outputs the ith bit of x for each $0 \leqq i < |x|$. It is clear that this condition can be tested in space polynomial in $2^n + |x| + |\pi|$, whence it follows easily that $\beta \in$ pspace.

Now fix one of the values of n computed by player II during a play (α, β) of the game, where α is an arbitrary strategy for player I. For each $|x| \leqq j \leqq 2^{n+1} - 1$, let total$(j)$ be the final value of total computed by VOTE during the while-loop cycle in which β decides s_j. (Here $|x|$ denotes the length of the original input to β and we insist that total$(2^{n+1} - 1)$ be defined even though the corresponding cycle of the while-loop is not actually executed.) Then β ensures that $0 \leqq \text{total}(|x|) < 2^{b \cdot 2^{n+1}}$ and $0 \leqq \text{total}(j+1) \leqq \frac{1}{2}\,\text{total}(j)$ for $|x| \leqq j < 2^{n+1} - 1$. It follows from these and the choice of n that $0 \leqq \text{total}(2^{n+1} - 1) < 1$, whence total$(2^{n+1} - 1) = 0$. But this implies that $KS[2^{cn}](R(\alpha, \beta)_{\leqq n}) \geqq b \cdot 2^{n+1}$. Since β establishes this condition for a different value of n during each of player II's turns, it follows that $R(\alpha, \beta) \notin X$. Thus β wins $G[X;$ all, pspace] for player II, so X is pspace-meager.

We now turn to the proof that X has pspace-measure 0. For each $j \in \mathbf{N}$, let X_j be the set of all languages L such that $KS[2^{cn}](L_{\leqq n}) < b \cdot 2^{n+1}$ holds for all $n \geqq \log j$. Clearly, $X = \cup\{X_j \mid j \in \mathbf{N}\}$. By Lemma 5.7, it suffices to show that this union is in fact a pspace-union of pspace-measure 0 sets.

Fix $j, k \in \mathbf{N}$ for a moment and choose the least $n \in \mathbf{N}$ such that $(1-b)2^{n+1} \geqq j + k + 2$. Let $\pi^{(0)}, \cdots, \pi^{(N-1)}$ be the lexicographic enumeration of all programs π of length $< b \cdot 2^{n+1}$ such that $U(\langle \pi, x \rangle)$ halts in $\leqq 2^{cn}$ space for all $|x| \leqq n$. Then it is easily checked that there is an enumerator $h_{jk} \in$ pspace such that

$$h_{jk}(0^l) = \begin{cases} U(\langle \pi^{(l)}, s_i \rangle)_{i=0}^{2^{n+1}-2} & \text{if } l < N \\ 0^{k+l+2} & \text{if } l \geqq N \end{cases}$$

holds for each $l \in \mathbf{N}$. In fact, since 2^n is linear in $j + k$, there is a function $h \in$ pspace such that this holds for all $j, k, l \in \mathbf{N}$. Similarly, there is a function $m \in$ pspace such that $m_{jk}(0^l) = 0^{N+l}$ for all $j, k, l \in \mathbf{N}$. It is now routine to check that each (h_{jk}, m_{jk}) is a pspace-cover of X_j with total measure $\mu^*(h_{jk}) \leqq 2^{-k}$. It follows that each (h_j, m_j) is a pspace-null cover of X_j, whence (h, m) testifies that X is a pspace-union of the pspace-measure 0 sets X_j. □

COROLLARY 6.2. *For any $c > 0$ and $b < 1$, the set of languages L with $KS[2^{cn}](L_{\leqq n}) \geqq b \cdot 2^{n+1}$ infinitely often is comeager and has measure* 1 *in* ESPACE.

If the game strategy β used in the first part of the proof of Theorem 6.1 is played against itself, then the result $R(\beta, \beta)$ is essentially the language constructed by Huynh

[1986b] in his proof of Theorem 1.2. In this sense, Theorem 1.2 is an immediate corollary of Theorem 6.1.

A simple modification of Theorem 6.1 and its proof gives the following result, which will be useful in § 7.

COROLLARY 6.3. *For any $c>0$ and $b<1$, the set of all languages L such that $KS[2^{cn}](L_{=n})<b\cdot 2^n$ almost everywhere is* pspace-*meager and has* pspace-*measure* 0. □

The situation in exponential time complexity classes is not as well understood as the situation in ESPACE. It is reasonable to conjecture that E contains languages whose KT-complexities are superpolynomial, but this implies $\mathrm{E}\not\subseteq \mathrm{P/Poly}$, which is a major open problem of complexity theory and cannot be proven by relativizable methods.

Here we prove a weaker analogue of Theorem 6.1. In order to formulate this result, we use the G-hierarchy of § 2 to define the following hierarchy of time-bounded Kolmogorov complexity classes.

DEFINITION 6.4. For each $i\geqq 1$,

$$\mathrm{KE}_i=\{L\mid KT[2^{G_{i-1}}](L_{\leqq n})\in G_{i-1}\}.$$

Each KE_i is an uncountable nonuniform complexity class. Nevertheless, these classes have the following useful properties.

LEMMA 6.5. *For each $i\geqq 1$, $\mathrm{E}_i\subseteq \mathrm{KE}_i\subseteq \mathrm{KE}_{i+1}$ and KE_{i+1} is closed under $\leqq_T^{\mathrm{P}_i}$-reductions and G_i closeness.*

Proof. It is obvious that $\mathrm{KE}_i\subseteq \mathrm{KE}_{i+1}$. If $L\in \mathrm{E}_i$, then $KT[2^{G_{i-1}}](L_{\leqq n})$ is bounded, so it is also clear that $\mathrm{E}_i\subseteq \mathrm{KE}_i$.

Assume $L'\leqq_T^{\mathrm{P}_i} L\in \mathrm{KE}_{i+1}$. Then there exist a G_i-time-bounded oracle machine M and a sequence $\pi^{(0)}, \pi^{(1)}, \cdots$ of programs such that $L'=L(M^L)$, $|\pi^{(n)}|\in \mathrm{G}_i$, and $U(\langle \pi^{(n)}, x\rangle)$ decides whether $x\in L$ in $\mathrm{G}_i(n)$ time for all $x\in\{0,1\}^{\leqq n}$ and $n\in\mathbf{N}$. For each n, then, consider a program $\hat{\pi}^{(n)}$ which simulates M, using $\pi^{(t(n))}$ to answer oracle queries, where $t\in \mathrm{G}_i$ is a bound on the running time of M. It is easily checked that the programs $\hat{\pi}^{(n)}$ testify that $L'\in \mathrm{KE}_{i+1}$. This proves that KE_{i+1} is closed under $\leqq_T^{\mathrm{P}_i}$-reductions.

Finally, assume L' is G_i close to KE_{i+1}, i.e., that $|(L'\triangle L)_{\leqq n}|\in \mathrm{G}_i$ for some $L\in \mathrm{KE}_{i+1}$. Then, since $i\geqq 1$, each $(L'\triangle L)_{\leqq n}$ has a listing whose length is G_i as a function of n. These listings can then be combined with programs testifying that $L\in \mathrm{KE}_{i+1}$ to get programs testifying that $L'\in \mathrm{KE}_{i+1}$. Thus KE_{i+1} is also closed under G_i closeness. □

We now prove a time analogue of Theorem 6.1. This says that most languages in E_{i+1} have high KT-complexity in the sense that they are not in KE_i.

THEOREM 6.6. *For $i\geqq 1$, KE_i is p_{i+1}-meager and has p_{i+1}-measure* 0.

Proof. Let X be the set of all languages L such that $KT[2^{\hat{g}_i(n)}](L_{\leqq n})<\hat{g}_i(n)$ almost everywhere. Then $\mathrm{KE}_i\subseteq X$, so it suffices to show that X is p_{i+1}-meager and has p_{i+1}-measure 0.

To see that X is p_{i+1}-meager, modify the strategy β used in the proof of Theorem 6.1 in the following ways.

(i) In the assignment to n, replace $(1-b)2^{n+1}$ with 2^n.

(ii) In VOTE, replace $b\cdot 2^{n+1}$ with $\hat{g}_i(n)$ and replace 2^{cn} space with $2^{\hat{g}_i(n)}$ time. Since $2^{\hat{g}_i(n)}=\hat{g}_{i+1}(2^n)$ and 2^n is linear in $|x|$, it is easy to check that the modified strategy β runs in G_{i+1} time, i.e., that $\beta\in \mathrm{p}_{i+1}$. Also, in a play (α,β) of the game, player II establishes the condition $0\leqq \mathrm{total}(2^{n+1}-1)<2^{\hat{g}_i(n)-2^n}$ for a different value of n during each of his turns. Since $\hat{g}_i(n)=o(2^n)$, it follows that $\mathrm{total}(2^{n+1}-1)=0$ for all sufficiently

large such n, whence $R(\alpha, \beta) \notin X$. That is, β wins $G[X; \text{all}, p_{i+1}]$ for player II, so X is p_{i+1}-meager by Theorem 4.3.

Now for each $j \in \mathbf{N}$, let X_j be the set of languages L such that $KT[2^{\hat{g}_i(n)}](L_{=n}) < \hat{g}_i(n)$ for all $n \geqq \log j$. We will show that X is a p_{i+1}-union of the p_{i+1}-measure 0 sets X_j.

Since $2^{\hat{g}_i(\log k)} \geqq \hat{g}_i(\hat{g}_i(\log k)) + k + 2$ holds for all but finitely many k, there is a finite modification g of $\hat{g}_i$ such that $g(k) \geqq k$ and $2^n \geqq \hat{g}_i(n) + k + 2$ hold for all k and all $n \geqq g(\log k)$.

Now, given j and k, we let $n = g(\log(j+k))$. This is easily computed and 2^n is G_{i+1} as a function of $j+k$.

There are fewer than $2^{\hat{g}_i(n)} = \hat{g}_{i+1}(2^n)$ programs π of length $< \hat{g}_i(n)$. The total time to run $U(\langle \pi, x \rangle)$ for $2^{\hat{g}_i(n)}$ steps for each such π and each $x \in \{0,1\}^{\leqq n}$ is thus G_{i+1} as a function of 2^n, hence as a function of $j+k$. Using this simulation, we can now imitate the second half of the proof of Theorem 6.1 to get $(h, m) \in p_{i+1}^2$ such that each (h_{jk}, m_{jk}) is a p_{i+1}-cover of X_j with total measure $\mu^*(h_{jk}) \leqq 2^{-k}$. This shows that X is a p_{i+1}-union of the p_{i+1}-measure 0 sets X_j, whence X has p_{i+1}-measure 0 by Lemma 5.7. □

COROLLARY 6.7. *For $i \geqq 1$, KE_i is meager and has measure 0 in E_{i+1}.*

The case $i = 2$ here says that most languages in E_3 have superpolynomial time-bounded Kolmogorov complexity.

7. Small circuits in exponential classes. Theorem 1.1 separates ESPACE from P/Poly ∩ ESPACE. In this section we widen this separation by proving that P/Poly is meager and has measure 0 in ESPACE. We then examine circuit-size complexity in exponential time complexity classes and prove, among other things, that P/Poly is meager and has measure 0 in E_3.

This investigation of small circuits in exponential complexity classes was our original motivation for the development of resource-bounded category and measure. Consequently, the main results of this section were originally proven directly. Here, however, we use Lemmas 7.1 and 7.5, both of which express a well-understood relationship between Kolmogorov complexity and circuit-size complexity, to easily derive the present results from those of § 6. For both of these lemmas, we fix a one-to-one *coding scheme*

$$\#: \{\text{circuits}\} \to \{0,1\}^*$$

and a constant $k_\# \in \mathbf{N}$ such that

(i) given $w, y \in \{0,1\}^*$, a deterministic TM can compute in polynomial time whether y is the code of a circuit c with $|w|$ inputs and, if so, the output of c on input w;

(ii) $|\#(c)| < k_\# \text{ size}(c) \log(n + \text{size}(c))$, where n is the number of inputs to c.

LEMMA 7.1. *If f is a nontrivial circuit-size bound, then every $L \in \text{SIZE}(f)$ has $KS[2^n](L_{=n}) < 2^{n-1}$ almost everywhere.*

Proof. Let f be such a bound and let

$$h(n) = k_\# g(n) \log(n + g(n)),$$

where g is a nontrivial circuit-size bound chosen so that $f = o(g)$. Note that $h(n) = o(2^n)$.

Now assume $L \in \text{SIZE}(f)$ and for each $n \in \mathbf{N}$, let c_n be a minimum-size circuit computing $L_{=n}$. Then for each n, we can combine a circuit-simulating machine M with the circuit code $\#(c_n)$ to get a program $\pi^{(n)}$ such that the following conditions hold for almost every n:

(i) $|\pi^{(n)}| < h(n) < 2^{n-1}$.

(ii) For each $x \in \{0,1\}^n$, $U(\langle \pi^{(n)}, x \rangle)$ correctly decides whether $x \in L$ in $\leqq 2^n$ space.

That is, the programs $\pi^{(n)}$ testify that $KS[2^n](L_{=n}) < 2^{n-1}$ almost everywhere. □

From Lemma 7.1 and Corollary 6.3, we immediately get the following theorem.

THEOREM 7.2. *If f is any nontrivial circuit-size bound, then* SIZE (f) *is* pspace-*meager and has* pspace-*measure* 0.

(The category portion of Theorem 7.2 was proven directly by a voting argument in Lutz [1987].)

COROLLARY 7.3. *If f is any nontrivial circuit-size bound, then* SIZE (f) *is meager and has measure* 0 *in* ESPACE.

COROLLARY 7.4. P/Poly *is meager and has measure* 0 *in* ESPACE.

We now turn to the matter of small circuits in exponential time complexity classes. Here it is convenient to use the KE-hierarchy introduced in § 6.

LEMMA 7.5. *If $i \geq 1$, then* SIZE $(G_i) \subseteq KE_{i+1}$.

Proof. Let f be an arbitrary circuit-size bound in G_i. Since $i \geq 1$, we can choose $g \in G_i$ such that $f = o(g)$. If we then define h from g as in the proof of Lemma 7.1, we will have $h \in G_i$ also.

Now assume $L \in$ SIZE (f). Then an easy modification of the argument used in the proof of Lemma 7.1 shows that $KT[G_i](L_{\leq n}) \in G_i$, whence $L \in KE_{i+1}$. Thus SIZE $(f) \subseteq KE_{i+1}$. □

From Lemma 7.5 and Theorem 6.6, the following theorem is immediate.

THEOREM 7.6. *If $i \geq 1$, then* SIZE (G_i) *is* p_{i+2}*-meager and has* p_{i+2}*-measure* 0.

COROLLARY 7.7. P/Poly *is meager and has measure* 0 *in* E_3.

If we fix a particular circuit-size bound $f \in G_i$ $(i \geq 1)$ and a language $L \in$ SIZE (f), then the proof of Lemma 7.5 gives us a function $g \in G_i$ such that $KT[g(n)](L_{\leq n}) < g(n)$ almost everywhere. It follows by the proof of Theorem 6.6 (with $\hat{g}_i$ replaced by g) that SIZE (f) is p_{i+1}-meager. This argument gives us the following corollaries. (Recall that $E_2 = EXP$.)

COROLLARY 7.8. *If $i \geq 1$ and $f \in G_i$, then* SIZE (f) *is* p_{i+1}*-meager and has* p_{i+1}*-measure* 0.

COROLLARY 7.9. *For each $k \in \mathbf{N}$,* SIZE (n^k) *is meager and has measure* 0 *in* E_2.

Since Wilson [1985] exhibits oracles under which $E_2 \subseteq$ P/Poly and $E \subseteq$ SIZE (n), Corollaries 7.7 and 7.9 take us about as far as we can go with relativizable techniques.

8. Information accessible by reducibilities. As mentioned in § 1, most intractability proofs for specific problems have taken the same form. Here we describe the more general *reducibility method*, which includes this form as a special case but may also lead to new lower bound arguments.

The reducibility method can be stated simply and informally as follows. Given a language L, let $\mathcal{R}(L)$ be the set of all languages that are efficiently reducible to L. Then the size of $\mathcal{R}(L)$, which is a measure of the amount of $\mathcal{R}$-accessible information in L, provides a lower bound for the complexity of computing L.

In applications of this method, which are taken here as paradigmatic, the size of $\mathcal{R}(L)$ is simply the matter of whether or not $\mathcal{R}(L)$ contains a particular complexity class $\mathcal{C}$, i.e., whether or not L is $\mathcal{R}$-hard for $\mathcal{C}$. To date, most uses of the reducibility method have followed this paradigm.

The recently proven Theorems 1.3 and 1.4 show that the paradigmatic reducibility method also gives lower bounds for "approximate recognition" of languages.

The primary weakness of the paradigm is its extremely primitive interpretation of the "size" of $\mathcal{R}(L)$. Unless L is $\mathcal{R}$-hard for $\mathcal{C}$, i.e., contains all information about $\mathcal{C}$ in $\mathcal{R}$-accessible form, the paradigm deems $\mathcal{R}(L)$ to be small and offers no nontrivial lower bound. Since most interesting intractable problems are probably not hard for

classes now known to contain intractability, this limitation is a severe one. It means, for example, that Theorems 1.3 and 1.4 are not likely to be applicable to interesting problems.

The remedy we propose is to use resource-bounded category and measure to refine this primitive notion of size. If we do this, then the reducibility method, as stated above, gives a quantitative relationship between the $\mathcal{R}$-accessible information content of L and the computational complexity of L.

The specific reducibilities of interest here are $\leqq_m^{\mathrm{P}}$, $\leqq_T^{\mathrm{P}}$, $\leqq_T^{\mathrm{PSPACE}}$, and $\leqq_T^{\mathrm{P}_i}$ $(i \geqq 1)$. It is thus convenient to define the set $\mathrm{P}_m(L) = \{L' \mid L' \leqq_m^{\mathrm{P}} L\}$, and to define the sets $\mathrm{P}_T(L)$, $\mathrm{PSPACE}_T(L)$, and $\mathrm{P}_{iT}(L)$ similarly from the other reducibilities.

Under the hypothesis that L is 2^{n^c} close to P for every $c > 0$, Theorems 1.3 and 1.4 tell us that $\mathrm{E} \not\subseteq \mathrm{P}_m(L)$ and $\mathrm{ESPACE} \not\subseteq \mathrm{P}_T(L)$, respectively. In §§ 9 and 10 we will show that this hypothesis in fact implies that $\mathrm{P}_m(L)$ is meager in E and that $\mathrm{PSPACE}_T(L)$ is meager and has measure 0 in ESPACE. That is, we replace conclusions of the form $\mathscr{C} \not\subseteq \mathcal{R}(L)$ with conclusions asserting that $\mathcal{R}(L) \cap \mathscr{C}$ is a very small subset of $\mathscr{C}$. Put differently, we replace conclusions stating that L does not contain all information about $\mathscr{C}$ in $\mathcal{R}$-accessible form with conclusions stating that L contains very little $\mathcal{R}$-accessible information about $\mathscr{C}$. Although these new conclusions appear to be considerably stronger, this has not been proven. We thus formulate the following hypotheses.

DEFINITION 8.1. If $\mathscr{C}$ is a complexity class and $\mathcal{R}$ is a reducibility, then the *category partial information hypothesis* for $\mathscr{C}$ and $\mathcal{R}$ is the assertion $\mathrm{PIH}_{\mathrm{category}}(\mathscr{C}, \mathcal{R})$, which says that there is a language $L \in \mathscr{C}$ such that $\mathcal{R}(L)$ does not contain $\mathscr{C}$ and $\mathcal{R}(L)$ is not meager in $\mathscr{C}$. The *measure partial information hypothesis* $\mathrm{PIH}_{\mathrm{measure}}(\mathscr{C}, \mathcal{R})$ is defined similarly, except that "meager" is replaced by "measure 0."

Thus partial information hypotheses assert the existence of languages containing "substantial but incomplete" information about $\mathscr{C}$ in $\mathcal{R}$-accessible form.

We make the following three conjectures.

CONJECTURE 8.2. $\mathrm{PIH}_{\mathrm{category}}(\mathrm{E}, \leqq_m^{\mathrm{P}})$ *holds.*

CONJECTURE 8.3. $\mathrm{PIH}_{\mathrm{category}}(\mathrm{ESPACE}, \leqq_T^{\mathrm{PSPACE}})$ *holds.*

CONJECTURE 8.4. $\mathrm{PIH}_{\mathrm{measure}}(\mathrm{ESPACE}, \leqq_T^{\mathrm{PSPACE}})$ *holds.*

If any of these conjectures hold, then the results of the following two sections do indeed increase the power of the reducibility method.

9. Information accessible in polynomial space. Theorem 1.4 says that, if L is 2^{n^c} close to P for every $c > 0$, then $\mathrm{P}_T(L)$ does not contain all of ESPACE. In this section we prove the stronger result that, if L is 2^{n^c} close to $\mathrm{DSPACE}(2^{n^c})$ for every $c > 0$, then $\mathrm{PSPACE}_T(L)$ is meager and has measure 0 in ESPACE. That is, a language that is approximable in feasible space does not contain significant polynomial-space-accessible information about ESPACE.

The key to Huynh's proof of Theorem 1.4 is Theorem 1.2, the existence of ESPACE languages with high space-bounded Kolmogorov complexity. Theorem 6.1, which says that most languages in ESPACE have this property, plays an analogous role in this section.

We first prove that almost all languages in ESPACE are very hard to approximate.

THEOREM 9.1. *If $c > 0$ and $b > 1$, then the set of languages that are $2^{n+1}/bn$ far from* $\mathrm{DSPACE}(2^{cn})$ *is* pspace-*comeager and has* pspace-*measure* 1.

Proof. Fix such c and b and suppose L is in the complement of this set, i.e., that there is an $O(2^{cn})$ space-bounded machine M such that $|(L \triangle L(M))_{\leqq n}| < 2^{n+1}/bn$ almost everywhere. Fix $0 < c' < c$ and $1/b < b' < 1$. Then for each n we can combine a

description of M with a listing of $(L \triangle L(M))_{\leqq n}$ to get a program $\pi^{(n)}$ such that the following conditions hold for almost all n.

(i) $|\pi^{(n)}| < b' \cdot 2^{n+1}$

(ii) $U(\langle \pi^{(n)}, s_i \rangle)_{i=0}^{2^{n+1}-2} = L_{\leqq n}$ in $\leqq 2^{c'n}$ space.

That is, the programs $\pi^{(n)}$ testify that $KS[2^{c'n}](L_{\leqq n}) < b' \cdot 2^{n+1}$ almost everywhere. By Theorem 6.1, the set of all L's with this property is pspace-meager and has pspace-measure 0. □

COROLLARY 9.2. *If $c > 0$ and $b > 1$, then the set of languages that are $2^{n+1}/bn$ far from* DSPACE (2^{cn}) *is comeager and has measure* 1 *in* ESPACE. □

We now give our improvement of Theorem 1.4.

THEOREM 9.3. *If L is 2^{n^c} close to* DSPACE (2^{n^c}) *for every $c > 0$, then* $\text{PSPACE}_T(L)$ *is* pspace-*meager and has* pspace-*measure* 0.

Proof. Let X be the set of languages L such that $KS[2^{n^c}](L_{\leqq n}) < 2^{n^c}$ holds almost everywhere for every $c > 0$. If the hypothesis holds, then an argument like that in the proof of Theorem 9.1 shows that $L \in X$. Since Theorem 6.1 says that X is pspace-meager and has pspace-measure 0, it thus suffices to prove that X is closed under $\leqq_T^{\text{PSPACE}}$.

Assume $L' \leqq_T^{\text{PSPACE}} L \in X$. Fix a $q(n)$-space-bounded oracle machine M such that $L' = L(M^L)$, where q is a polynomial. Also, for each $c > 0$ and $n \in \mathbf{N}$, fix a program $\pi(c, n)$ testifying to the value of $KS[2^{n^c}](L_{\leqq n})$.

Now for each c and n, consider a program $\pi'(c, n)$ which simulates M, using $\pi(c, q(n))$ to answer oracle queries. Then there is a constant $d > 0$, not depending on c or n, such that for almost all c, for almost all n, the following conditions hold.

(i) $|\pi'(c, n)| \leqq 2^{q(n)^c} + d$.

(ii) For all $x \in \{0, 1\}^{\leqq n}$, $U(\langle \pi'(c, n), x \rangle)$ decides whether $x \in L'$ in $\leqq q(n) + 2^{q(n)^c} + d$ space.

Now let $c_1 > 0$ be arbitrary and choose $c > 0$ such that $q(n) + 2^{q(n)^c} < 2^{n^{c_1}}$ almost everywhere. Then the programs $\pi'(c, n)$ testify that $KS[2^{n^{c_1}}](L'_{\leqq n}) < 2^{n^{c_1}}$ almost everywhere. Thus $L' \in X$, whence X is indeed closed under $\leqq_T^{\text{PSPACE}}$ and the proof is complete. □

COROLLARY 9.4. *If L is 2^{n^c} close to* DSPACE (2^{n^c}) *for every $c > 0$, then* $\text{PSPACE}_T(L)$ *is meager and has measure* 0 *in* ESPACE.

Thus, if a language can be shown to contain significant polynomial-space-accessible information about ESPACE, it will follow that the language is not very close to PSPACE.

10. Information accessible in polynomial time. If Conjecture 8.3 or Conjecture 8.4 holds, then § 9 already extends the class of languages that can be proven intractable by the reducibility method. However, any language that is susceptible to the method of § 9 still must lie "well outside" of PSPACE. Since most languages that we would like to prove intractable are elements of PSPACE, it follows that we need a finer method, i.e., a method that applies to a finer reducibility.

Theorem 1.3 says that, if L is 2^{n^c} close to P for every $c > 0$, then $\text{P}_m(L)$ does not contain all of E. In this section we show that, if L is 2^{n^c} close to DTIME (2^{n^c}) for every $c > 0$, then $\text{P}_m(L)$ is actually meager in E. That is, a language that is approximable in feasible time contains only meager $\leqq_m^{\text{P}}$-accessible information about E. If Conjecture 8.2 holds, this strengthens Theorem 1.3.

The basis of § 9 is Theorem 6.1, which says that most ESPACE languages are incompressible in a space-bounded algorithmic sense. Similarly, the present section is based on Theorem 10.2 below, a technical result which says, in part, that most languages in E are incompressible in a time-bounded, many-one sense. The following definition,

which specifies this sense, uses the function $m_g(n) = |\{x \in \{0,1\}^{\leqq n} \mid \exists y \in \{0,1\}^{\leqq n}[x \neq y \text{ and } g(x) = g(y)]\}|$ to quantify the rate at which a function $g : \{0,1\}^* \to \{0,1\}^*$ fails to be one-to-one.

DEFINITION 10.1. (1) A language L is *$f(n)$-incompressible* by $\leqq^{\mathrm{P}}_m$-reductions if every $\leqq^{\mathrm{P}}_m$-reduction g of L has $m_g(n) \leqq f(n)$ for infinitely many n. If every $\leqq^{\mathrm{P}}_m$-reduction g of L is one-to-one almost everywhere (i.e., if every such m_g is bounded), then L is *strongly incompressible* by $\leqq^{\mathrm{P}}_m$-reductions.

(2) A language L is *simultaneously $f(n)$-incompressible* by $\leqq^{\mathrm{P}}_m$-reductions *and $h(n)$ far* from a set X of languages if for each $\leqq^{\mathrm{P}}_m$-reduction g of L and each language $L' \in X$ there are infinitely many n for which $m_g(n) \leqq f(n)$ and $|(L \triangle L')_{\leqq n}| \geqq h(n)$ both hold.

THEOREM 10.2. *For any $\varepsilon > 0$, any $a > 0$, and any nondecreasing, unbounded function $f(n)$ that is computable in $2^{O(n)}$ time, the set of all languages that are simultaneously $2^{\varepsilon n}$-incompressible by $\leqq^{\mathrm{P}}_m$-reductions and $2^{n+1}/f(n)$ far from* DTIME(2^{an}) *is* p*-comeager.*

Proof. Let $X = X(\varepsilon, a, f)$ be the set of all such languages, where we assume without loss of generality that $\varepsilon \leqq 1$ and $f(n) \leqq 2^n$ for all n. Let $M_0, M_1, \cdots$ and $T_0, T_1, \cdots$ be standard enumerations of the Turing machine acceptors and transducers, respectively.

If T_k is a transducer and $x, y \in \{0,1\}^*$, we say that $T_k(x) = T_k(y)$ *in time t* if $T_k(x) = T_k(y)$ and T_k halts in $\leqq t$ steps on each of the inputs x and y. We then say that x *defies* T_k if there exist $i < j < |x|$ such that the ith and jth bits of x are different but $T_k(s_i) = T_k(s_j)$ in time $|x|$.

If x defies T_k and $x \sqsubseteq L$, it is clear that T_k is not a many-one reduction of L. It is also clear that the predicate "x defies T_k" can be evaluated in time polynomial in $|x|$.

Now let $\Theta(k, i, j, x, n)$ be a predicate asserting that $k < |x|$, x does not defy T_k, $i < j$, $|x| \leqq j < 2^{n+1} - 1$, and $T_k(s_i) = T_k(s_j)$ in time 2^{n+1}. Note that the condition $(\exists k, i, j)\Theta(k, i, j, x, n)$ can be tested in time polynomial in $|x| + 2^n$. Consider the strategy $\beta = \beta(\varepsilon, a, f)$ defined as follows.

begin
  input x;
  $z, n, \ell := x, \lceil \varepsilon^{-1} \log(2 + |x|) \rceil, 0$;
  if $(\exists k, i, j)\Theta(k, i, j, x, n)$
    then fix such i, j with k minimum
    else $i, j := 0, 1$;
  while $|z| < 2^{n+1} - 1$ **do**
  cases
    $|z| = i$: $z := z0$
    $|z| = j$: $z := zb$, where b is the negation of the ith bit of z
    else: **if** M_ℓ accepts $s_{|z|}$ in $\leqq 2^{(a+1)n}$ time
      then $z, \ell := z0, (\ell + 1) \bmod \lfloor f(n)/4 \rfloor$
      else $z, \ell := z1, (\ell + 1) \bmod \lfloor f(n)/4 \rfloor$
  end cases and while;
  output z
end β.

Since 2^n is polynomial in $|x|$ here, it is easy to check that $\beta \in$ p. In fact, we will show here that β is a winning strategy for player II in $\mathrm{G}[X^c; \text{all}, \text{p}]$, where $X^c = \mathscr{P}(\{0,1\}^*) \setminus X(\varepsilon, a, f)$. To this end, let $L = R(\alpha, \beta)$, where α is an arbitrary strategy for player I. It suffices to show that $L \in X$.

Fix a $\leqq_m^P$-reduction g of L and a language $L' \in \mathrm{DTIME}(2^{an})$, with witnesses T_k and M_ℓ, respectively.

Since $g = T_k$ is a reduction of L, no initial segment of L defies T_k. This implies that the first if-test in β is not true with k as the least witness during any move by player II. Since T_k runs in polynomial time, this in turn implies that, for all but finitely many of player II's moves, $m_g(n) \leqq |x| < 2^{\varepsilon n}$.

The machine M_ℓ runs in $O(2^{an})$ time on any input in $\{0,1\}^{\leqq n}$, so the while-loop in β ensures that, for all but finitely many of player II's moves, $|(L \triangle L')_{\leqq n}| \geqq \lfloor (2^{n+1} - |x| - 3)/\lfloor f(n)/4 \rfloor \rfloor \geqq (2^{n+3} - 4|x| - 12)/f(n) - 1 \geqq (2^{n+2} - 12)/f(n) \geqq 2^{n+1}/f(n)$. Since g and L' are arbitrary here, we have now shown that $L \in X$. Thus β does indeed win $G[X^c; \text{all}, p]$ for player II, so Theorem 4.3 tells us that X is p-co-meager. □

COROLLARY 10.3. *For any $\varepsilon > 0$, any $a > 0$, and any nondecreasing, unbounded function $f(n)$ that is computable in $2^{O(n)}$ time, the set of all languages that are simultaneously $2^{\varepsilon n}$-incompressible by $\leqq_m^P$-reductions and $2^{n+1}/f(n)$ far from* DTIME (2^{an}) *is comeager in* E.

If the game strategy β used in the above proof is played against itself, where $\varepsilon = c = 1$ and $f(n) = n$, then we get the following result, which is the basis of Schöning's proof of Theorem 1.3.

COROLLARY 10.4. *There is a language $L \in$ E which is strongly incompressible by $\leqq_m^P$-reductions and $2^n/n$ far from* P.

If we ignore the incompressibility in Theorem 10.2, then we immediately get the following.

THEOREM 10.5. *If $c > 0$ and $f(n)$ is any nondecreasing, unbounded function that is computable in $2^{O(n)}$ time, then the set of languages that are $2^{n+1}/f(n)$ far from* DTIME (2^{cn}) *is comeager in* E. □

Since $f(n)$ may be an extremely slow-growing function, this is a very strong nonapproximability theorem. It says that, in the sense of category, most languages in E cannot be approximated with an error rate that converges to 0 in any feasible way.

We finally come to our improvement of Theorem 1.3.

THEOREM 10.6. *If L is 2^{n^c} close to* DTIME (2^{n^c}) *for every $c > 0$, then* $P_m(L)$ *is* p-*meager.*

Proof. Assume that $P_m(L)$ is not p-meager and let X be the set of languages in Theorem 10.2, where $\varepsilon = \frac{1}{2}$, $a = 1$, and $f(n) = n$. Since X is p-comeager, there is a language $A \in X \cap P_m(L)$. Fix such, let g be a $\leqq_m^P$-reduction of A to L, let q be a polynomial such that $|g(x)| \leqq q(|x|)$ for all x, and choose $0 < c < b$ such that $q(n)^b \leqq n$ almost everywhere. We will show that L is 2^{n^c} far from DTIME (2^{n^c}).

Let $L' \in \mathrm{DTIME}(2^{n^c})$. Then $g^{-1}(L') \in \mathrm{DTIME}(2^n)$ and $A \in X$, so there exist infinitely many n such that $m_g(n) \leqq 2^{n/2}$ and $|(A \triangle g^{-1}(L'))_{\leqq n}| \geqq 2^{n+1}/n$. For any sufficiently large such n we thus have $|(L \triangle L')_{\leqq q(n)}| \geqq (2^{n+1}/n) - 2^{n/2} \geqq 2^n/n \geqq 2^{n^{c/b}} \geqq 2^{q(n)^c}$. Since $L' \in \mathrm{DTIME}(2^{n^c})$ is arbitrary here, it follows that L is 2^{n^c} far from DTIME (2^{n^c}). □

COROLLARY 10.7. *If L is 2^{n^c} close to* DTIME (2^{n^c}) *for every $c > 0$, then* $P_m(L)$ *is meager in* E.

Thus, if a language L can be shown to contain nonmeager $\leqq_m^P$-accessible information about E, it will follow that the language is not very close to P. In fact, the proofs of Theorems 10.2 and 10.6 show that this will follow if it is just shown that player II does not have a winning strategy for the game $G[P_m(L); \text{all}, p]$.

It is not known whether Theorem 10.6 holds with $P_T(L)$ in place of $P_m(L)$ or with "measure 0" in place of "meager," but it is now easy to get the following much weaker result.

THEOREM 10.8. *If* $i \geqq 1$ *and* L *is* G_i *close to* E_{i+1}, *then* $P_{iT}(L)$ *is* p_{i+2}*-meager and has* p_{i+2}*-measure* 0.

Proof. By Lemma 6.5, the hypothesis implies that $P_{iT}(L) \subseteq KE_{i+1}$, so this follows immediately from Theorem 6.6. □

COROLLARY 10.9. *If* L *is polynomially close to* E_2, *then* $P_T(L)$ *is meager and has measure* 0 *in* E_3.

11. Conclusion. Resource-bounded category and measure have been introduced and shown to reveal new structure in many complexity classes. This structure has been used to refine known relationships between uniform and nonuniform complexity measures. It has also been used as the basis for a new formulation of the reducibility method.

The important open questions here concern the partial information hypotheses. If any of Conjectures 8.2, 8.3, or 8.4 hold, then the newly formulated reducibility method is indeed more powerful than the old one. Of course it would be ideal for these conjectures to be shown to hold with interesting, natural problems as witnesses, as the work here then gives lower bounds for such problems.

In any case, it is already clear that resource-bounded category and measure interact in interesting ways with resource-bounded reducibilities, nonuniform complexity measures, approximation, and other much studied structural aspects of complexity classes. It is expected that the study of such interactions will continue to yield clarifying insights.

Acknowledgment. I am very much indebted to Alexander Kechris and Yaser Abu-Mostafa, my major and minor advisors, respectively. I thank them very gratefully for many helpful discussions and for the constant support, encouragement, and guidance I have received from them. I also thank Giora Slutzki and an anonymous referee for useful observations.

REFERENCES

J. L. BALCÁZAR, R. V. BOOK, AND U. SCHÖNING [1986], *Sparse sets, lowness, and highness*, SIAM J. Comput., 15, pp. 739–747.

S. A. COOK [1971], *The complexity of theorem proving procedures*, in Proceedings of the 3rd ACM Symposium on Theory of Computing, Shaker Heights, OH, pp. 151–158.

R. I. FREIDZON [1972], *Families of recursive predicates of measure zero*, translated in J. Soviet Math., 6 (1976), 4, pp. 449–455.

P. R. HALMOS [1950], *Measure Theory*, Springer-Verlag, Berlin, New York.

J. HARTMANIS [1983], *Generalized Kolmogorov complexity and the structure of feasible computations*, in Proceedings of the 24th IEEE Symposium on Foundations of Computer Science, pp. 439–445.

J. HARTMANIS AND Y. YESHA [1984], *Computation times of* NP *sets of different densities*, Theoret. Comput. Sci., 34, pp. 17–32.

D. T. HUYNH [1986a], *Some observations about the randomness of hard problems*, SIAM J. Comput., 15, pp. 1101–1105.

——— [1986b], *Resource-bounded Kolmogorov complexity of hard languages*, in Structure in Complexity Theory, Lecture Notes in Computer Science, Springer-Verlag, Berlin, New York, Vol. 223, pp. 184–195.

R. KANNAN [1982], *Circuit-size lower bounds and non-reducibility to sparse sets*, Inform. and Control, 55, pp. 40–56.

R. M. KARP [1972], *Reducibility among combinatorial problems*, in Complexity of Computer Computations, R. E. Miller and J. W. Thatcher, eds., Plenum Press, New York, pp. 85–104.

R. M. KARP AND R. J. LIPTON [1980], *Some connections between nonuniform and uniform complexity classes*, in Proceedings of the 12th ACM Symposium on Theory of Computing, Los Angeles, CA, pp. 302–309.

A. N. KOLMOGOROV [1933], *Foundations of the Theory of Probability*, English translation, Chelsea Publishing, New York, 1950.

L. A. LEVIN [1973], *Universal sorting problems*, in Problems of Information Transmission, 9, Consultants Bureau, New York, London, pp. 265-266.

L. R. LISAGOR [1979], *The Banach-Mazur game*, translated in Math. USSR Sbornik, 38 (1981), 2, pp. 201-206.

O. B. LUPANOV [1958], *On the synthesis of contact networks*, Dokl. Akad. Nauk. SSSR, 119, pp. 23-26.

J. H. LUTZ [1987], *Resource-bounded Baire category and small circuits in exponential space*, Structure in Complexity Theory, Proceedings of Second Annual Conference, IEEE Computer Society Press, New York, pp. 81-91.

K. MEHLHORN [1973], *On the size of sets of computable functions*, in Proceedings of the 14th IEEE Symposium on Switching and Automata Theory, Iowa City, IA, pp. 190-196.

——— [1974], *The "almost all" theory of subrecursive degrees is decidable*, in Proceedings of the Second Colloquium on Automata, Languages and Programming, Springer Lecture Notes, Springer-Verlag, Berlin, New York, pp. 317-325.

A. R. MEYER AND L. STOCKMEYER [1972], *The equivalence problem for regular expressions with squaring requires exponential space*, in Proceedings of the 13th IEEE Symposium on Switching and Automata Theory, Iowa City, IA, pp. 125-129.

J. C. OXTOBY [1971], *Measure and Category*, Springer-Verlag, Berlin, New York.

U. SCHÖNING [1983], *A low and a high hierarchy within* NP, J. Comput. System Sci., 27, pp. 14-28.

——— [1986], *Complete sets and closeness to complexity classes*, Math. Systems Theory, 19, pp. 29-41.

C. E. SHANNON [1949], *The synthesis of two-terminal switching circuits*, Bell Systems Technical Journal, 28, pp. 59-98.

M. SIPSER [1983], *A complexity-theoretic approach to randomness*, in Proceedings of the 15th ACM Symposium on Theory of Computing, Boston, MA, pp. 330-335.

L. STOCKMEYER AND A. K. CHANDRA [1979], *Provably difficult combinatorial games*, SIAM J. Comput., 8, pp. 151-174.

C. B. WILSON [1985], *Relativized circuit complexity*, J. Comput. System Sci., 31, pp. 169-181.

Y. YESHA [1983], *On certain polynomial-time truth-table reducibilities of complete sets to sparse sets*, SIAM J. Comput., 12, pp. 411-425.

SIAM J. COMPUT.
Vol. 19, No. 6, pp. 1132-1161, December 1990

011

COMPUTING PUISEUX-SERIES SOLUTIONS TO DETERMINANTAL EQUATIONS VIA COMBINATORIAL RELAXATION*

KAZUO MUROTA†

Abstract. Let $A(t, x) = (A_{ij}(t, x))$ be a square matrix with A_{ij} being a polynomial in t and x. This paper proposes an algorithm for computing the Puiseux (= fractional power) series solutions $x = x(t)$ to the equation $\det A(t, x) = 0$. The algorithm is based on an observation which links the Newton diagram (polygon) for $\det A(t, x)$ with the perfect matchings of a bipartite graph associated with A. The algorithm is efficient, making full use of available fast network-type algorithms.

Key words. computer algebra, determinant, Newton diagram (polygon), Puiseux-series expansion, combinatorial optimization, convex hull, parametric assignment problem, sensitivity of eigenvalues

AMS(MOS) subject classifications. 68Q40, 05C50, 15A15, 68Q25

1. Introduction.

Let $A(t, x) = (A_{ij}(t, x))$ be an $n \times n$ matrix with

$$A_{ij}(t, x) = \sum_{s \in \mathbf{Z}} \sum_{r \in \mathbf{Q}} A_{ijrs} t^r x^s, \tag{1}$$

where the coefficients A_{ijrs} are elements of a certain field, and the summations are assumed to involve a finite number of terms.

We are interested in the computational procedure for the Puiseux (= fractional power) series solutions $x = x(t)$ to the equation $\det A(t, x) = 0$. This problem arises in many different contexts; we may be interested in the sensitivity of the (generalized) eigenvalue x of the matrix A which is subject to a perturbation t. In fact, the author was motivated by a problem of this type which appeared in the sensitivity analysis of bifurcation phenomena of truss structures (Ikeda and Murota [16]).

If we could explicitly compute the expansion

$$f(t, x) = \det A(t, x) = \sum_{s} \sum_{r} f_{rs} t^r x^s \tag{2}$$

to find $r(s) = \min \{r \mid f_{rs} \neq 0\}$ for each s, then we could apply the standard procedure using the Newton diagram (polygon) that computes the Puiseux-series solutions of an algebraic equation (see § 2.1 for details). This method plots all the points $(s, r(s))$ on a plane and considers the convex hull (or convex epigraph, to be more precise) of those points; the slope p of the sides of this convex polygon gives the order of the first term $x \sim \gamma t^{-p}$ in the Puiseux-series solutions.

The present work is motivated by the following observations:

(1) Even when the matrix A is moderately sized, say, $n = 10$, explicit enumeration of all the nonzero coefficients f_{rs} would be prohibitively time- (and memory-) consuming.

(2) The exponent p is determined by those pairs (s, r) which correspond to the extreme points (or vertices) of the convex hull. The other points (s, r) lying inside contribute nothing, and therefore, need not be computed.

* Received by the editors September 20, 1989; accepted for publication (in revised form) April 23, 1990.

† Department of Mathematical Engineering and Information Physics, University of Tokyo, Bunkyo-ku, Tokyo 113, Japan. This work was done while the author stayed at the Institute of Econometrics and Operations Research, University of Bonn, Federal Republic of Germany. The author was supported by the Alexander von Humboldt Foundation.

(3) There is a close relation between the nonzero terms f_{rs} and the perfect matchings of a bipartite graph associated with A. So long as no accidental numerical cancellation occurs in the determinant expansion, the extreme points (s, r) can be identified by solving a parametric assignment (or weighted bipartite matching) problem (see § 3 for the details).

The present work is an attempt to establish a link between computer algebra [3], [9], [24] and mathematical programming (combinatorial optimization [18], in particular). We make use of results in mathematical programming in two different ways. First, the proposed algorithm uses the results from network flow theory in its individual steps; the correctness relies on the duality theorem and the practical efficiency on the fast network-type algorithms. Second, the whole algorithm is designed in line with some general methods known in mathematical programming. In particular, we make use of the idea of "relaxation" (and "cutting plane") which typically appears in integer programming [25], and also the idea of "artificial variable" which appears in the simplex method in linear programming [5], [8].

The basic idea of the proposed algorithm may be described in general terms as follows. As mentioned above, the exponent in $x \sim \gamma t^{-p}$ is determined from the extreme points of

$$N(A) = \{(s, r) \mid f_{rs} \neq 0\}.$$

Instead of working directly with $N(A)$, we consider its combinatorial counterpart $\hat{N}(A)$ based on the relation between the nonzero terms in $\det A$ and the perfect matchings in a bipartite graph associated with A. The combinatorial counterpart $\hat{N}(A)$, to be called the "combinatorial relaxation" to $N(A)$, has the properties that $\hat{N}(A) \supseteq N(A)$ and that $\hat{N}(A) = N(A)$ if no numerical cancellation occurs in the determinant expansion of A. The combinatorial relaxation $\hat{N}(A)$ has the computational advantage that the extreme points can found by efficient network-type algorithms.

In our algorithm we first solve the easier problem defined by the relaxation $\hat{N}(A)$, hoping that the solution thus obtained is also valid for the original problem defined by $N(A)$. If the solution is not good for $N(A)$, we modify the matrix A slightly, so that $N(A)$ is kept invariant and at the same time the invalid solution is eliminated from the relaxation. In summary, the proposed algorithm consists of the following three phases:

Phase 1. Finding a solution to the relaxation.

Phase 2. Testing for the validity of this solution in the original problem.

Phase 3 (In case of invalid solution). Modifying the relaxation so that the invalid solution is eliminated.

It would be worthwhile comparing our present approach to the cutting-plane method in integer programming, as follows. To be specific, consider an integer program:

$$\text{Minimize} \quad cx \qquad \text{subject to} \quad x \in S,$$

where

$$S = S(A) = \{x \mid Ax \geqq b, x \in \mathbf{Z}^n\}.$$

The cutting-plane method first solves its relaxation to a linear program:

$$\text{Minimize} \quad cx \qquad \text{subject to} \quad x \in \hat{S},$$

where

$$\hat{S} = \hat{S}(A) = \{x \mid Ax \geqq b\},$$

and then tests for the integrality of the solution $\hat{x}$ of the relaxation; in case $\hat{x}$ is not integral, it modifies A by introducing an additional constraint (cutting plane) in such a way that $\hat{x}$ no longer belongs to $\hat{S}(A)$ and $S(A)$ is not changed. The three general phases mentioned above should be evident in this context.

The outline of this paper is as follows. In § 2 some fundamental results on the Newton diagram and the weighted bipartite matching are described. In § 3 the key concept of "combinatorial relaxation" to the Newton diagram is introduced. In § 4 the basic approach of the proposed algorithm is explained, while the major steps of the algorithm are described in §§ 5–8, and the whole algorithm is shown in § 9. The termination and the complexity of the algorithm are discussed in §10. Notation is listed in the Appendix.

2. Preliminaries.

2.1. Newton diagram. Consider a polynomial in x and in a fractional power of t:

$$f(t, x) = \sum_{s \in \mathbf{Z}} \sum_{r \in \mathbf{Q}} f_{rs} t^r x^s, \tag{3}$$

where the coefficients f_{rs} are elements of a certain field, and the summations are assumed to involve a finite number of terms.

We describe the method of a Newton diagram (polygon) for obtaining the solution $x = x(t)$ to the equation $f(t, x) = 0$ in the form of (formal) Puiseux series (or fractional power series with bounded denominators of the fractional exponents) in t:

$$x(t) = \gamma_1 t^{-p_1} + \gamma_2 t^{-p_1 - p_2} + \cdots, \tag{4}$$

where $\gamma_i \neq 0$, $p_i \in \mathbf{Q}$ for $i = 1, 2, \cdots$, and $p_i < 0$ for $i \geqq 2$. Note that $f(t, x) = 0$ is an algebraic equation in x and can have as many solutions $x = x(t)$ as its degree in x; in particular, p_1 is not uniquely determined.

The exponent p_1 can be found with the aid of the Newton diagram:

$$N = \{(s, r) \mid f_{rs} \neq 0\}.$$

We say that a line l *supports* N, if all the points in N lie above (or on) l and if $l \cap N \neq \emptyset$. To be more precise, let $r = \alpha s + \beta$ denote the equation for l. Then l supports N if $r \geqq \alpha s + \beta$ for all $(s, r) \in N$ and if equality holds for some $(s, r) \in N$. Note that $\beta = \min \{r - \alpha s \mid (s, r) \in N\}$. Let us say in this paper that l *tightly supports* N, if, in addition, $|l \cap N| \geqq 2$. We also define the *southwest point* $SW(N) = (s^*, r^*)$ of N by

$$s^* = \min \{s \mid (s, r) \in N\}, \qquad r^* = \min \{r \mid (s^*, r) \in N\}. \tag{5}$$

It is not difficult to see that p_1 is given by the slope α of a line: $r = \alpha s + \beta$, which tightly supports N. The coefficient γ_1 is determined from the leading terms in the equation

$$f(t, \gamma_1 t^{-p_1}) = t^{\beta} \sum_{r - p_1 s = \beta} f_{rs} \gamma_1^s + o(t^{\beta}) = 0. \tag{6}$$

Once the leading term $x(t) \sim \gamma_1 t^{-p_1}$ is found, the second term is determined by applying the above procedure to the equation

$$f(t, t^{-p_1}(\gamma_1 + x)) = 0. \tag{7}$$

From the computational-theoretical point of view, however, this recursive procedure is not free from tough complexity issues associated with approximating γ_1. In this paper we do not enter into these issues, concentrating on the computation of the first-order term.

Remark 2.1. Evidently it is sufficient to consider $\{(s, r(s))\}$ instead of N, where $r(s) = \min\{r \mid f_{rs} \neq 0\}$. Then the above procedure is applicable to more general algebraic equations in x with coefficients being Puiseux series in t. In fact, the Newton diagram means more often the diagram consisting of $\{(s, r(s))\}$ in this general case. It may also be mentioned that the equation $f(t, x) = 0$ (even in the general case) is known to have a (formal) Puiseux-series solution if the underlying field is algebraically closed with characteristic zero (e.g., **C**) (see, e.g., [2], [4], [11], [15], [26]).

Remark 2.2. Finding the convex hull of a planar point set is one of the most fundamental problems in computational geometry. A number of efficient algorithms are known for this problem (see, e.g., [23]).

Example 2.1. Consider

$$f(t, x) = -t + (t^2 + t^4)x^2 - t^6x^4.$$

The Newton diagram N is shown in Fig. 1, where $SW(N) = (0, 1)$. This shows two possibilities for p_1, namely, $p_1^{(1)} = \frac{1}{2}$ and $p_1^{(2)} = 2$. The corresponding coefficients γ_1 are determined as $\gamma_1^{(1)} = \pm 1$ and $\gamma_1^{(2)} = \pm 1$, respectively, from

$$f(t, \gamma_1 t^{-1/2}) = -t(1 - \gamma_1^2) + o(t)$$

and

$$f(t, \gamma_1 t^{-2}) = t^{-2}\gamma_1^2(1 - \gamma_1^2) + o(t^{-2}).$$

Thus we obtain four solutions: $x \sim \pm t^{-1/2}, \pm t^{-2}$.

2.2. Assignment problem. Let G be a bipartite graph with vertex set $V = R \cup C$ and edge set E; we assume that $|R| = |C|$. The initial and the terminal vertex of edge $e \in E$ are denoted by $\partial^+ e$ and $\partial^- e$; all the edges are directed from R to C so that $\partial^+ e \in R$ and $\partial^- e \in C$. Parallel edges are allowed, i.e., there can be two distinct edges e and e' such that $\partial^+ e = \partial^+ e'$ and $\partial^- e = \partial^- e'$. A matching is a subset M of E such that $|M| = |\partial^+ M| = |\partial^- M|$, where $\partial^+ M = \{\partial^+ e \mid e \in M\}$, etc., and a perfect matching (or an assignment) is a matching M with $|M| = |R|$.

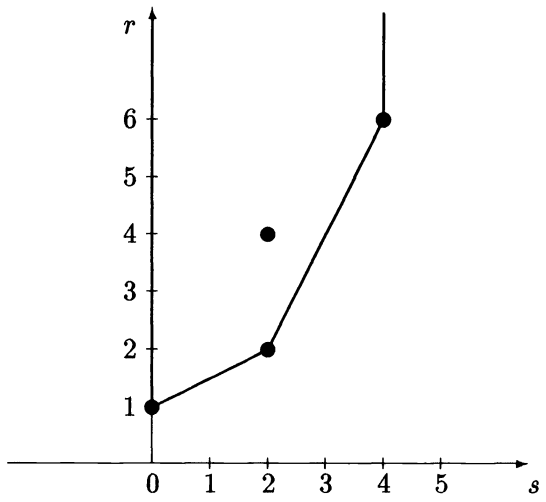


FIG. 1. *Newton diagram N of Example* 2.1.

Suppose a cost c_e is given for each edge e, i.e., $c: E \to \mathbf{R}$. The cost of a matching M is defined by

$$c(M) = \sum_{e \in M} c_e.$$

The assignment problem is to find a perfect matching with minimum/maximum cost. The optimality of a perfect matching is expressed in terms of the *potentials* (or *dual variables*) each associated with a vertex of G. Namely, a perfect matching M has the minimum cost if and only if there exist potentials u_i^R $(i \in R)$ and u_j^C $(j \in C)$ such that

$$\begin{aligned} &c_e + u_i^R - u_j^C \geqq 0, \quad i = \partial^+ e, \quad j = \partial^- e \quad \forall e \in E, \\ &c_e + u_i^R - u_j^C = 0, \quad i = \partial^+ e, \quad j = \partial^- e \quad \forall e \in M. \end{aligned} \tag{8}$$

Similarly, a perfect matching M has the maximum cost if and only if there exist potentials v_i^R $(i \in R)$ and v_j^C $(j \in C)$ such that

$$\begin{aligned} &c_e - v_i^R + v_j^C \leqq 0 \qquad i = \partial^+ e, \quad j = \partial^- e \quad \forall e \in E, \\ &c_e - v_i^R + v_j^C = 0 \qquad i = \partial^+ e, \quad j = \partial^- e \quad \forall e \in M. \end{aligned} \tag{9}$$

We may assume that the potentials are integers if the costs are integers. We may also assume that potentials are chosen to be (dual) basic, in correspondence to a tree in G. This guarantees that the difference of potentials at two vertices is bounded by

$$(2n-1) \max \{|c_e| \mid e \in E\}. \tag{10}$$

There are a number of very efficient algorithms for finding the optimal assignment as well as the associated potentials. See, e.g., [5] and [18] for more about the assignment problem.

3. Combinatorial relaxation of the Newton diagram. This section introduces the key concept of combinatorial relaxation to the Newton diagram for $\det A$. This concept will enforce the link between linear algebra (determinant) and graph theory (matching) observed in various contexts [19], [20]. First recall that we have defined

$$N(A) = \{(s, r) \mid f_{rs} \neq 0\}$$

with reference to (1) and (2). We call $N(A)$ the *Newton diagram* for A.

The structure of the matrix A of (1) is conveniently represented by the bipartite graph $G = G(A) = G(A; p)$ defined as follows. The vertex set $V = V(G)$ is the disjoint union of the row set R and the column set C of A. The edge set $E = E(G)$ is identified with the nonzero terms in the entries of A, i.e.,

$$E = \{(ijrs) \mid A_{ijrs} \neq 0\},$$

where

$$\partial^+(ijrs) = i \in R, \qquad \partial^-(ijrs) = j \in C.$$

Note that G has parallel edges from $i \in R$ to $j \in C$ if $A_{ij}(t, x)$ contains more than one term. To edge $(ijrs)$ is given a cost parametrized by p:

$$c_{ijrs}(p) = r - ps.$$

By considering the expansion of the determinant, we see that

$$\det A(t, x) = \sum_M A_M t^{r(M)} x^{s(M)}, \tag{11}$$

where the summation is taken over all perfect matchings M in $G(A)$, and

$$A_M = \pm \prod_{(ijrs)\in M} A_{ijrs} \qquad (\neq 0),$$

$$r(M) = \sum_{(ijrs)\in M} r, \qquad s(M) = \sum_{(ijrs)\in M} s.$$

Suggested by this expression, we define

$$\hat{N}(A) = \{(s(M), r(M)) \mid M : \text{perfect matching in } G(A)\}, \tag{12}$$

and name it the *combinatorial Newton diagram* or the *combinatorial relaxation* to $N(A)$. It should be emphasized that $\hat{N}(A)$ is a combinatorial notion in the sense that the numerical values of the coefficients A_{ijrs} are disregarded.

These two notions, $N(A)$ and $\hat{N}(A)$, are closely related as follows. Putting

$$\mathcal{M}(s, r) = \{M \mid (s(M), r(M)) = (s, r)\},$$

we obtain from (11)

$$\det A(t, x) = \sum_{(s,r)} \left(\sum_{M\in\mathcal{M}(s,r)} A_M \right) t^r x^s. \tag{13}$$

Comparing this with (2), we see

$$(s, r) \in \hat{N}(A) \quad \text{if } (s, r) \in N(A),$$

and that the converse is also true unless $\sum \{A_M \mid M \in \mathcal{M}(s, r)\}$ is equal to zero due to "accidental" numerical cancellation. Such numerical cancellation does not occur, if, e.g., the nonzero coefficients A_{ijrs} are algebraically independent over some base field. Hence we obtain the following statements, which would justify the name of "relaxation" for $\hat{N}(A)$.

PROPOSITION 3.1. (1) $N(A) \subseteq \hat{N}(A)$.

(2) $N(A) = \hat{N}(A)$ *if the nonzero coefficients* A_{ijrs} *are algebraically independent.*

We call $(s, r) \in \hat{N}(A)$ *genuine* if $(s, r) \in N(A)$, and *spurious* if $(s, r) \notin N(A)$. We also say that a supporting (respectively, tightly supporting) line of $\hat{N}(A)$ is *genuine* or *spurious* according as it is a supporting (respectively, tightly supporting) line of $N(A)$ or not. Proposition 3.1(1) implies that a supporting (respectively, tightly supporting) line l of $\hat{N}(A)$ is genuine if $l \cap N(A) \neq \emptyset$ (respectively, if $|l \cap N(A)| \geqq 2$).

The tightly supporting lines of $\hat{N}(A)$ can be computed efficiently on the basis of the following relation to the parametric assignment problem on $G(A) = G(A; p)$. Let $c(p)$ denote the minimum cost of a perfect matching in $G(A; p)$ with respect to the parametrized cost $c_{ijrs} = r - ps$. The minimum cost $c(p)$ is a concave piecewise-linear function in p. The *breakpoints* are defined as those points at which the slope of $c(p)$ changes. Then, we have the following obvious but important statement, which is a consequence of the well-known point-line duality.

PROPOSITION 3.2. p *is a breakpoint of* $G(A; p)$ *if and only if* p *is the slope of a tightly supporting line of* $\hat{N}(A)$.

Efficient combinatorial algorithms for finding breakpoints are available (see Remark 9.1 in § 9).

Example 3.1. Consider a 3×3 matrix

$$A(t, x) = \begin{pmatrix} 1 & 0 & t^2x^2 \\ 1+tx & t & t^3x^3 \\ t^3x^2 & 1+t^2 & -1 \end{pmatrix}, \tag{14}$$

for which we have

$$\det A = -t + (t^2 + t^4)x^2 - t^6x^4 + \text{spur}\,(t^3x^3, t^5x^3),$$

where spur $(\cdots)$ is the list of terms that appear in (11) and are cancelled out. The graph $G(A)$ is illustrated in Fig. 2, in which the cost is attached to edges. The combinatorial Newton diagram $\hat{N}(A)$ is shown in Fig. 3, in which the genuine points are indicated by solid disks (●) and the spurious points by open circles (○). Note that $G(A)$ has eight perfect matchings, among which two cancelling pairs yield the spurious points $(s, r) = (3, 3)$ and $(3, 5)$. $\hat{N}(A)$ has three tightly supporting lines with slopes $p = \frac{1}{2}, 1, 3$. As we have seen in Example 2.1, only the first is genuine, corresponding to the solution $x \sim \pm t^{-1/2}$.

4. Outline of the algorithm. In this section we give an intuitive description of the main idea of the proposed algorithm for determining all possible first-order approximations $x \sim \gamma t^{-p}$ to the solution of $\det A(t, x) = 0$. The complete description will be given in § 9.

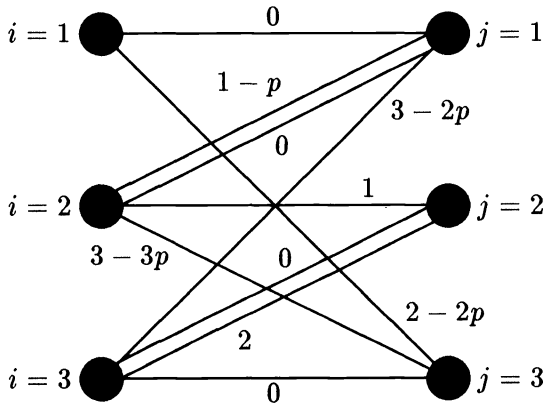


FIG. 2. *Graph $G(A)$ of Example* 3.1.

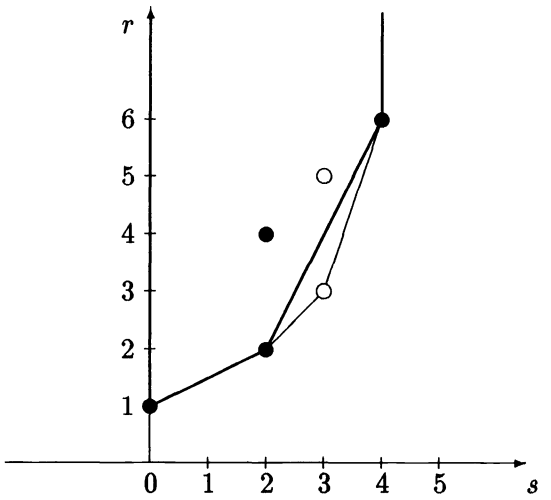


FIG. 3. *Combinatorial Newton diagram $\hat{N}(A)$ of Example* 3.1.

Recall (cf. § 2.1) that the order of magnitude $-p$ is determined by the geometrical configuration of $N(A)$, whereas the coefficient γ is to be computed numerically. The main idea of the algorithm lies in the following observations.

(1) The p is determined from the tightly supporting lines of $N(A)$ (cf. § 2.1).

(2) $N(A)$ is approximated by $\hat{N}(A)$ (cf. Proposition 3.1).

(3) Tightly supporting lines of $\hat{N}(A)$ can be computed by efficient combinatorial algorithms without constructing $\hat{N}(A)$ explicitly (cf. Proposition 3.2, Remark 9.1).

As we have seen in Example 3.1, however, not all tightly supporting lines of $\hat{N}(A)$ correspond to the solution of $\det A = 0$. Namely, the genuine lines give the solutions while the spurious ones do not. To cope with the spurious lines we take note of the fact that (rowwise) elimination operations on A can modify $\hat{N}(A)$ without affecting $N(A)$. We will show how to modify or adjust A by elimination operations so that the particular spurious point in question may be eliminated from $\hat{N}(A)$ (for the adjusted A). It should be emphasized that the proposed modification of A by elimination is only local and computationally efficient. Furthermore, this procedure is invoked, hopefully, only rarely, i.e., only when the numerical cancellation results in a spurious extreme point.

In our algorithm we maintain a genuine supporting line l for $\hat{N}(A)$ and a point $P \in l \cap N(A)$. Recall (cf. § 3) that l is a genuine supporting line for $\hat{N}(A)$ if and only if l supports $\hat{N}(A)$ and $l \cap N(A) \neq \emptyset$. If we have $|l \cap N(A)| \geqq 2$ for such l, then l is a tightly supporting line for $N(A)$, and hence the slope p of l corresponds to a desired solution.

The prototype of the algorithm is described below in geometric terms. Note that the slope p of l is nondecreasing in the course of the algorithm. It should be emphasized that we never generate all the points of $\hat{N}(A)$. See Example 4.1 below for illustration.

ALGORITHM (outline).

Step 1 (initial point). Find a genuine supporting line l for $\hat{N}(A)$ and a point $P \in l \cap N(A)$ such that $p \approx -\infty$ (i.e., the slope p of l is sufficiently small).

Step 2 (solution to relaxation). Rotate l counterclockwise around P so that l tightly supports $\hat{N}(A)$;
If l is vertical, i.e., parallel to the r-axis, stop;
Let P' be the rightmost point (i.e., with the largest s-coordinate) of $l \cap \hat{N}(A)$.

Step 3.1 (genuine P'). If $P' \in N(A)$, then the slope p of l corresponds to a solution $x \sim \gamma t^{-p}$;
Determine the coefficient γ numerically;
$P := P'$; Go to Step 2.

Step 3.2 (spurious P'). If $P' \notin N(A)$ then modify A by rowwise eliminations so that $P' \notin \hat{N}(A)$, $N(A)$ is not changed and l is a supporting line for $\hat{N}(A)$; Go to Step 2.

The reader is advised to put this algorithm into the general framework of relaxation explained in the Introduction and, in particular, to identify the three phases mentioned there.

In order to implement the above algorithm we need algorithms for the following subproblems.

SUBPROBLEM 1 (initial point). To find the starting pair (l, P) in Step 1.

SUBPROBLEM 2 (solution to relaxation). To find the tightly supporting line for $\hat{N}(A)$ with the next larger slope in Step 2.

SUBPROBLEM 3 (test for membership in $N(A)$). To test in Step 3 whether P' of $\hat{N}(A)$ is genuine or spurious.

SUBPROBLEM 4 (modification of A). To find the adjustment scheme of A in Step 3.2.

Our approach to these problems is briefly described here.

Subproblem 1 (initial point). The southwest point $SW(\hat{N}(A)) = (s^*, r^*)$, where (s^*, r^*) is defined by (5) with N replaced by $\hat{N}(A)$, can be computed as the solution to the assignment problem on $G(A; p)$ with $p \approx -\infty$ (sufficiently small). Usually, the southwest point may be expected to be genuine and then it serves as the initial P; any line l passing through P with the slope $p \approx -\infty$ will do. If, unfortunately, the southwest point $SW(\hat{N}(A))$ is spurious, we mimic the starting procedure of "artificial variables" in the (two-phase) simplex method for linear programming (see, e.g., [5], [8]). Namely, we modify the matrix A by introducing a number of "artificial terms" along the diagonal. The modified problem has the genuine southwest point and therefore the algorithm can be started. After a number of steps, we will find that the artificial terms play no roles, then we are solving the original problem. The details are described in § 8.

Subproblem 2 (solution to relaxation). As already discussed in § 3, the tightly supporting lines of $\hat{N}(A)$ are in one-to-one correspondence with the breakpoints of the parametric assignment problem on $G(A; p)$ with cost $c_{ijrs}(p) = r - ps$. Hence finding the tightly supporting line with the next larger slope amounts to finding the next larger breakpoint of $G(A; p)$. For the latter problem, a number of efficient combinatorial algorithms are available. See Remark 9.1.

Subproblem 3 (test for membership in $N(A)$). It is all-important to note that only extreme points of $\hat{N}(A)$ are to be tested for membership in $N(A)$. An extreme point $P' = (s, r)$ of $\hat{N}(A)$ corresponds to the minimum assignment on $G(A; p)$ for some p. In the notation of (13), P' is genuine if and only if

$$\sum_{M \in \mathcal{M}(s,r)} A_M \neq 0.$$

With the help of potentials (= dual variables) (cf. § 2.2) we can extract those terms in $A(t, x)$ which can contribute to this sum; i.e., those terms for which (8) holds with strict inequality cannot contribute to this sum, and therefore may be discarded. In this way the membership test for an extreme point $P' \in \hat{N}(A)$ is reduced to the test for nonsingularity of a numerical matrix composed of part of the coefficients A_{ijrs}. The details are described in § 6.

Subproblem 4 (modification of A). It is not trivial to modify A with a small amount of computation so that P' is eliminated from $\hat{N}(A)$ while maintaining the condition that l should support $\hat{N}(A)$. Again the potentials associated with assignment problems play substantial roles both to extract those terms of $A(t, x)$ which contribute to P' and to maintain the condition above. An additional annoying phenomenon is that the elimination operation on A can give rise to new spurious points, which may force the algorithm to run forever in the loop of Step 2 and Step 3.2. Therefore some special care must be taken in the elimination operations for guaranteed finite termination of the algorithm. The elimination scheme is given in § 7 and the termination and the complexity are discussed in § 10.

Example 4.1. The above algorithm is applied to the problem of Example 3.1. The flow of computation is traced below. Recall Fig. 3.

Step 1. Fortunately the southwest point $P = (0, 1)$ is genuine and we can start with this P. The line l is: $r = ps + 1$ with $p \approx -\infty$.

Step 2. l is now rotated around $P = (0, 1)$ to contain a side of $\hat{N}(A)$, so that we have l: $r = \frac{1}{2}s + 1$.
$P' := (2, 2)$.

Step 3.1. Since $P'=(2,2)\in N(A)$, we obtain one solution with $p=\frac{1}{2}$.
The coefficient $\gamma=\pm 1$ is obtained as the solution $x=\gamma$ to the equation $\det D(x)=x^2-1=0$, where

$$D(x)=\begin{pmatrix}1 & 0 & x^2\\ 1 & 1 & 0\\ 0 & 1 & -1\end{pmatrix}.$$

(This matrix is extracted from the original A with the aid of the potentials. Thus we have obtained the first set of solutions $x\sim\pm t^{-1/2}$.)
$P:=(2,2)$.

Step 2. l is rotated around $P=(2,2)$ to contain another side of $\hat{N}(A)$, and we have l: $r=s$.
$P':=(3,3)$.

Step 3.2. Since $P'=(3,3)\notin N(A)$, we modify A by $A:=W(t,x)A$ with

$$W(t,x)=\begin{pmatrix}1 & -t^{-1}x^{-1} & 0\\ 0 & 1 & 0\\ 0 & 0 & 1\end{pmatrix}$$

to obtain

$$A=\begin{pmatrix}-t^{-1}x^{-1} & -x^{-1} & 0\\ 1+tx & t & t^3x^3\\ t^3x^2 & 1+t^2 & -1\end{pmatrix}. \tag{15}$$

(The combinatorial Newton diagram $\hat{N}(A)$ is now changed to that in Fig. 4, whereas $N(A)$ is invariant since A is modified through a unit triangular matrix $W(t,x)$. Note that $P'=(3,3)\notin\hat{N}(A)$ and a new spurious point $(s,r)=(-1,0)$ has appeared. The current line l: $r=s$ is still a genuine supporting line.)

Step 2. l is rotated counterclockwise around $P=(2,2)$ to contain a side of $\hat{N}(A)$, and we have l: $r=2s-2$.
$P':=(4,6)$.

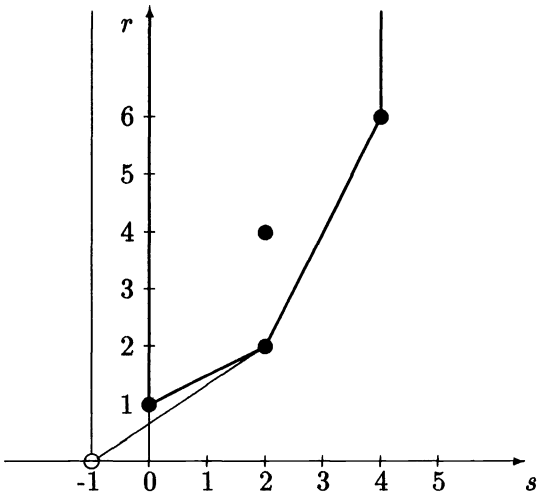


FIG. 4. $\hat{N}(A)$ *of Example* 4.1 (*with modified* A).

Step 3.1. Since $P' = (4, 6) \in N(A)$, we obtain another solution with $p = 2$.
The coefficient $\gamma = \pm 1$ is obtained as the solution $x = \gamma$ $(\neq 0)$ to the equation $\det D(x) = -x^2 - x^4 = 0$, where

$$D(x) = \begin{pmatrix} -x^{-1} & -x^{-1} & 0 \\ 0 & 0 & x^3 \\ x^2 & 1 & -1 \end{pmatrix}.$$

(We have obtained the second set of solutions $x \sim \pm t^{-2}$.)
$P := (4, 6)$.

Step 2. l is rotated to a vertical line, and we stop the algorithm.

In this illustration we have used two matrices, $W(t, x)$ and $D(x)$, which have not been defined before, and will be considered in §§ 6 and 7. At this point, however, it is noted that the entries of $W(t, x)$ and $D(x)$ are monomials (possibly with negative exponents), respectively, in (t, x) and in x. Moreover, $W(t, x)$ can be expressed as a product of diagonal scaling factors and a constant matrix; i.e., the matrix $W(t, x)$ above can be written as

$$W(t, x) = \operatorname{diag}(1, tx, t^{-2}x^{-2}) \cdot \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \cdot \operatorname{diag}(1, t^{-1}x^{-1}, t^2x^2). \qquad \square$$

5. Useful transformations. The following three kinds of transformations for the matrix $A(t, x)$ are used, explicitly or implicitly, throughout this paper. For a vector $u = (u_i \mid i = 1, \cdots, n)$ and a variable x in general, we put

$$\operatorname{diag}(x; u) = \operatorname{diag}(x^{u_1}, x^{u_2}, \cdots, x^{u_n})$$

for notational convenience.

The first is a transformation of A by means of row/column scaling by diagonal matrices having monomial entries (possibly with negative exponents):

$$A'(t, x) = \operatorname{diag}(t; r^R) \cdot \operatorname{diag}(x; s^R) \cdot A(t, x) \cdot \operatorname{diag}(x; -s^C) \cdot \operatorname{diag}(t; -r^C),$$

where

$$s^R = (s_i^R \mid i = 1, \cdots, n), \qquad s^C = (s_j^C \mid j = 1, \cdots, n),$$
$$r^R = (r_i^R \mid i = 1, \cdots, n), \qquad r^C = (r_j^C \mid j = 1, \cdots, n).$$

This transformation has no essential effect on $N(A)$ and $\hat{N}(A)$ in that it causes a mere coordinate shift in the (s, r)-plane, namely,

$$(s, r) \in N(A) \Leftrightarrow (s + \Delta s, r + \Delta r) \in N(A'),$$
$$(s, r) \in \hat{N}(A) \Leftrightarrow (s + \Delta s, r + \Delta r) \in \hat{N}(A'),$$

where

$$\Delta s = \sum_i s_i^R - \sum_j s_j^C, \qquad \Delta r = \sum_i r_i^R - \sum_j r_j^C.$$

The variables $\{s_i^R, s_j^C\}$ and $\{r_i^R, r_j^C\}$ may be regarded as the potentials associated with the assignment problem in $G(A)$.

The second transformation is a change of the dependent variable

$$A'(t, x) = A(t, t^{-p}x).$$

This corresponds to another coordinate change for $N(A)$ and $\hat{N}(A)$, namely,

$$(s, r) \in N(A) \Leftrightarrow (s, r - ps) \in N(A'),$$

$$(s, r) \in \hat{N}(A) \Leftrightarrow (s, r - ps) \in \hat{N}(A').$$

Note that the r-axis is invariant.

The last kind of transformation is a rowwise (sweeping-out) operation by a constant nonsingular matrix W:

$$A'(t, x) = W \cdot A(t, x).$$

This has an essential effect on $\hat{N}(A)$ and keeps $N(A)$ invariant. Often W is a triangular matrix (with respect to some orderings of rows and columns).

6. Testing for membership in $N(A)$. This section affords an algorithm for testing for the membership in $N(A)$ of an extreme point of $\hat{N}(A)$ (cf. Subproblem 3 in § 4).

6.1. Degree of determinant. Let $D(x) = (D_{ij}(x))$ be an $n \times n$ matrix with

$$D_{ij}(x) = \sum_{s \in \mathbf{Z}} D_{ijs} x^s,$$

where the summation is assumed to be a finite sum. We denote by $\delta(D)$ (or $\delta_x(D)$) the maximum degree of a nonzero term in $\det D(x)$:

$$\delta(D) = \deg_x \det D(x),$$

where $\delta(D)$ may possibly be negative. We often use the term "degree" in this extended sense.

A bipartite graph $G^* = G^*(D)$ is associated with $D(x)$ in a similar manner as $G(A)$ is with $A(t, x)$. The vertex set $V(G^*)$ is the disjoint union of the row set R and the column set C of D, and the edge set $E(G^*)$ is identified with the nonzero entries of D, i.e.,

$$E(G^*) = \{(ij) \mid D_{ij}(x) \neq 0\}.$$

Note that, unlike $G(A)$, G^* has no parallel edges. To edge (ij) is attached a cost:

$$c_{ij} = \max \{s \mid D_{ijs} \neq 0\} = \deg_x D_{ij}(x).$$

We define $\hat{\delta}(D)$ to be the maximum cost of a perfect matching in $G^*(D)$.

The argument in § 3 shows the following.

PROPOSITION 6.1. (1) $\delta(D) \leqq \hat{\delta}(D)$.

(2) $\delta(D) = \hat{\delta}(D)$ *if the nonzero coefficients* D_{ijs} *are algebraically independent.*

We say that $D(x)$ is *upper tight* (or u-*tight*) if $\delta(D) = \hat{\delta}(D)$.

A procedure [21] is given below which tests for u-tightness of $D(x)$ without computing all terms of $\det D(x)$. Let v_i^R $(i \in R)$ and v_j^C $(j \in C)$ be the potentials associated with a maximum perfect matching that satisfy (9) (cf. § 2.2). Then

$$\hat{\delta}(D) = \Delta v,$$

where

$$\Delta v = \sum_i v_i^R - \sum_j v_j^C, \tag{16}$$

and

$$\tilde{c}_{ij} \equiv c_{ij} - v_i^R + v_j^C \leqq 0. \tag{17}$$

Consider a perfect matching M in $G^*(D)$. It follows from (17) that M has the maximum cost $\hat{\delta}(D) = \Delta v$ if and only if (17) holds with equality for all $(ij) \in M$. In other words, a term with strict inequality in (17) can never contribute to a maximum matching and may be deleted without any influence on the coefficient of $x^{\hat{\delta}(D)}$ in the determinant expansion.

Define $\tilde{D}(x) = (\tilde{D}_{ij}(x))$ by

$$\tilde{D}_{ij}(x) = \begin{cases} D_{ijc_{ij}} x^{c_{ij}} & \text{if } \tilde{c}_{ij} = 0, \\ 0 & \text{otherwise} \end{cases}$$

and $D^* = (D^*_{ij})$ by

$$D^*_{ij} = \begin{cases} D_{ijc_{ij}} & \text{if } \tilde{c}_{ij} = 0, \\ 0 & \text{otherwise.} \end{cases} \tag{18}$$

Note that

$$\tilde{D}(x) = \operatorname{diag}(x; v^R) \cdot D^* \cdot \operatorname{diag}(x; -v^C),$$

and hence $D^* = \tilde{D}(1)$. The following proposition shows that the test for u-tightness of $D(x)$ is reduced to the test for nonsingularity of a constant matrix D^*.

PROPOSITION 6.2. *$D(x)$ is* u*-tight if and only if D^* is nonsingular.*

Proof. Consider a perfect matching M in $G^*(D)$. In the determinant expansion of $D(x)$, this matching corresponds to

$$\prod_{(ij) \in M} D_{ij}(x),$$

which yields terms with degree less than or equal to

$$c(M) = \sum_{(ij) \in M} c_{ij}.$$

We see by (17) that $c(M) = \hat{\delta}(D)$ if and only if $\tilde{c}_{ij} = 0$ for all $(ij) \in M$. Hence

$$\det D(x) = \det \tilde{D}(x) + o(x^{\hat{\delta}(D)}),$$

where the last term means an expression consisting of terms with degree strictly less than $\hat{\delta}(D)$. Finally, we note that

$$\det \tilde{D}(x) = (\det D^*) x^{\hat{\delta}(D)},$$

completing the proof. □

6.2. Testing for membership in $N(A)$. Let $P' = (s', r')$ be an extreme point (of the convex epigraph) of $\hat{N}(A)$. Then there exists a supporting line l of $\hat{N}(A)$ such that $P' \in l \cap \hat{N}(A)$. We assume that P' is the rightmost point in $l \cap \hat{N}(A)$, since this is the case in Step 2 of our algorithm (§ 4).

We can naturally translate these geometrical statements into the language of the assignment problem on $G(A; p)$ as follows. Since $P' \in \hat{N}(A)$ is an extreme point, there exists a closed interval $[p^{(1)}, p^{(2)}]$ of parameter values of p and a perfect matching M in $G(A; p)$ such that M has the minimum cost for $p \in [p^{(1)}, p^{(2)}]$ and $(s', r') = (s(M), r(M))$. Let $u_i^R = u_i^R(p)$ and $u_j^C = u_j^C(p)$ be the associated potentials satisfying (8); we have

$$\tilde{c}_{ijrs} \equiv r - ps + u_i^R - u_j^C \geqq 0, \tag{19}$$

and the equality holds for $(ijrs) \in M$. Then the line l defined by

$$r = ps - \Delta u \tag{20}$$

supports $\hat{N}(A)$ at P', where

$$\Delta u = \sum_i u_i^R - \sum_j u_j^C. \tag{21}$$

Furthermore, P' is the rightmost point of $l \cap \hat{N}(A)$ if and only if $p < p^{(2)}$.

Let us extract from $A(t, x)$ those terms which can contribute to the minimum assignment in $G(A)$. Define $D(x) = (D_{ij}(x))$ by

$$D_{ij}(x) = \sum \{A_{ijrs} x^s \mid (s, r) \in E_{ij}\}, \tag{22}$$

where

$$E_{ij} = \{(s, r) \mid \tilde{c}_{ijrs} = 0, (ijrs) \in E(G)\}, \tag{23}$$

and consider the bipartite graph $G^*(D)$ associated with $D(x)$ as in § 6.1. It follows from (19) that a perfect matching M' in $G(A)$ has the minimum cost if and only if (19) holds with equality for all $(ijrs) \in M'$. In view of this and (20) we see that a perfect matching in $G^*(D)$ corresponds to a perfect matching M' in $G(A)$ such that $(s(M'), r(M')) \in l$, and, conversely, a perfect matching M' in $G(A)$ with $(s(M'), r(M')) = (s', r')$ has a corresponding perfect matching in $G^*(D)$. In particular, the s-coordinate of P' is given (cf. (16) for notation) by

$$s' = \hat{\delta}(D) = \Delta v. \tag{24}$$

The following characterizations provide us with a computational procedure to test for the membership of $P' \in \hat{N}(A)$ in $N(A)$.

PROPOSITION 6.3. *For the rightmost point P' of $l \cap \hat{N}(A)$, the following three statements are equivalent, where $D(x)$ is defined by* (22) *and D^* by* (18):

(1) *P' is genuine (i.e., $P' \in N(A)$).*

(2) *$D(x)$ is* u*-tight (i.e., $\delta(D) = \hat{\delta}(D)$).*

(3) *D^* is nonsingular.*

Proof. The equivalence of (1) and (2) follows from the relation between matchings on $G(A)$ and on $G^*(D)$ explained above. The equivalence of (2) and (3) is already given in Proposition 6.2. □

Example 6.1. Let A be the matrix of (14) used in Examples 3.1 and 4.1. The line l (with slope $p = 1$) defined by $r = s$ tightly supports $\hat{N}(A)$ and $P' = (3, 3)$ is the rightmost point of $l \cap \hat{N}(A)$ (see Fig. 3). As the potentials satisfying (19) we may choose $u_i^R = u_j^C = 0$ $(i, j = 1, 2, 3)$. According to (22) we have

$$D(x) = \begin{pmatrix} 1 & 0 & x^2 \\ 1+x & 0 & x^3 \\ 0 & 1 & -1 \end{pmatrix}. \tag{25}$$

The associated bipartite graph $G^*(D)$, illustrated in Fig. 5, has two perfect matchings both corresponding to $P' = (3, 3)$. As the potentials satisfying (17) we may choose

$$v_1^R = 0, \quad v_2^R = 1, \quad v_3^R = -2,$$
$$v_1^C = 0, \quad v_2^C = -2, \quad v_3^C = -2.$$

This shows that $\hat{\delta}(D) = \Delta v = 3$. According to (18) we have

$$D^* = \begin{pmatrix} 1 & 0 & 1 \\ 1 & 0 & 1 \\ 0 & 1 & -1 \end{pmatrix}. \tag{26}$$

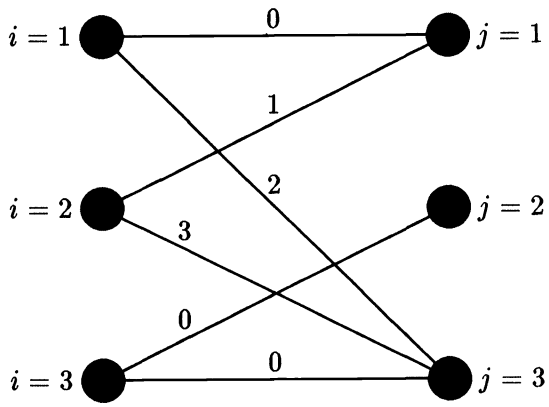


FIG. 5. *Graph $G^*(D)$ of Example* 6.1.

Since D^* is singular, Proposition 6.3 shows that $P'=(3,3)\notin N(A)$, which is also equivalent to $\hat{\delta}(D)\,(=3)>\delta(D)\,(=2)$. This is how we have found that $P'=(3,3)\notin N(A)$ in Example 4.1, Step 3.2.

7. Modification of matrix. This section deals with the modification of A in Step 3.2 of our algorithm when the point $P'\in \hat{N}(A)$ turns out to be spurious (cf. Subproblem 4 in § 4). To be more precise, we are given a tightly supporting line l with slope p of $\hat{N}(A)$ such that $l\cap N(A)\neq\emptyset$ and that the rightmost point $P'\in l\cap\hat{N}(A)$ does not belong to $N(A)$. We are to modify $A(t,x)$ to another matrix $A'(t,x)=(A'_{ij}(t,x))$ such that

(P1) $N(A')=N(A)$.

(P2) l supports $\hat{N}(A')$.

(P3) The rightmost point of $l\cap\hat{N}(A')$ lies to the left of P', i.e., has strictly smaller s-coordinate than that of P'.

Note that the last condition implies, in particular, that $P'\notin\hat{N}(A')$. We also require the following additional properties:

(P4) $\delta^*(A')\leqq\delta^*(A)$, where

$$\delta^*(A)=\max\{\deg_x A_{ij}(t,x)\,|\,i\in R, j\in C\}.$$

(P5) If $A(t,x)$ contains only integer powers of t and x, so does $A'(t,x)$.

Recall from § 6 that a matrix $D(x)$ is associated with P' by (22) and that a constant matrix D^* is derived from $D(x)$ with reference to the potentials $\{v_i^R, v_j^C\}$. We have term-rank $D^*=n$ since $P'\in\hat{N}(A)$ and rank $D^*<n$ since $P'\notin N(A)$ (cf. Proposition 6.3), where term-rank of a matrix means the maximum size of a matching in the bipartite graph associated with the matrix as in § 6.1. Assume that W is a nonsingular constant matrix such that

$$\text{term-rank}\,(WD^*)<n. \tag{27}$$

Using the potentials $\{u_i^R, u_j^C\}$ for $G(A;p)$ and $\{v_i^R, v_j^C\}$ for $G^*(D)$, we define

$$s_{ij}=v_i^R-v_j^C,\qquad r_{ij}=ps_{ij}-u_i^R+u_j^C,$$

and put

$$\sigma(i,k) = s_{ij} - s_{kj} = v_i^R - v_k^R,$$
$$\rho(i,k) = r_{ij} - r_{kj} = p(v_i^R - v_k^R) - (u_i^R - u_k^R).$$

We define the transformation from A to A' by

$$\begin{aligned} A'_{ij}(t,x) &= \sum_k t^{\rho(i,k)} x^{\sigma(i,k)} W_{ik} A_{kj}(t,x) \\ &= \sum_k \sum_s \sum_r W_{ik} A_{kjrs} t^{r+\rho(i,k)} x^{s+\sigma(i,k)}, \end{aligned} \tag{28}$$

i.e.,

$$\begin{aligned} A'(t,x) = \operatorname{diag}(t; -u^R + pv^R) \cdot \operatorname{diag}(x; v^R) \cdot W \\ \cdot \operatorname{diag}(x; -v^R) \cdot \operatorname{diag}(t; u^R - pv^R) \cdot A(t,x). \end{aligned} \tag{29}$$

The second expression reveals that

$$\det A'(t,x) = \det W \cdot \det A(t,x),$$

which implies the property (P1). We claim that the properties (P2) and (P3) are also satisfied.

PROPOSITION 7.1. *Properties* (P2) *and* (P3) *hold if* W *satisfies* (27).

Proof. (P2). For any i, j, k, r, s such that $W_{ik}A_{kjrs} \neq 0$ in (28), we have

$$(r + \rho(i,k)) - p(s + \sigma(i,k)) + u_i^R - u_j^C = r - ps + u_k^R - u_j^C = \tilde{c}_{kjrs} \geqq 0, \tag{30}$$

where the last inequality is due to (19).

Let $(\hat{s}, \hat{r})$ be any point in $\hat{N}(A')$ and consider a corresponding perfect matching M in $G(A')$. Then

$$\hat{s} = \sum_{(ijrs) \in M} (s + \sigma(i,k)), \qquad \hat{r} = \sum_{(ijrs) \in M} (r + \rho(i,k)),$$

where k denotes an index which varies with $(ijrs)$. Since l is given by the equation (20): $r = ps - \Delta u$, the above inequality (30) implies that $(\hat{s}, \hat{r})$ lies above (or on) l. Noting $l \cap N(A') = l \cap N(A) \neq \emptyset$, we conclude that l supports $\hat{N}(A')$.

(P3) As we have done in § 6.2, we can throw away some terms of $A'(t,x)$ in considering the points of $l \cap \hat{N}(A')$. In parallel to (22) we define $D'(x) = (D'_{ij}(x))$ by

$$D'_{ij}(x) = \sum_k W_{ik} \sum \{A_{kjrs} x^{s+\sigma(i,k)} \mid (s,r) \in E_{kj}\}.$$

Using the expression $\sigma(i,k) = v_i^R - v_k^R$, we may rewrite this as

$$D'_{ij}(x) = x^{v_i^R - v_j^C} \sum_k W_{ik} \sum \{A_{kjrs} x^{s - v_k^R + v_j^C} \mid (s,r) \in E_{kj}\}.$$

By (17) the last term can be expressed as

$$\sum \{A_{kjrs} x^{s - v_k^R + v_j^C} \mid (s,r) \in E_{kj}\} = D^*_{kj} + O\left(\frac{1}{x}\right),$$

where $O(1/x)$ means an expression consisting of terms with negative powers of x. Therefore,

$$D'_{ij}(x) = x^{v_i^R - v_j^C} \left(\sum_k W_{ik} D^*_{kj} + O\left(\frac{1}{x}\right) \right),$$

i.e.,

$$D'(x) = \mathrm{diag}\,(x; v^R) \cdot \left(WD^* + O\left(\frac{1}{x}\right) \right) \cdot \mathrm{diag}\,(x; -v^C).$$

Since term-rank $(WD^*) < n$ by (27), we see

$$\hat{\delta}(D') \leqq \Delta v - 1,$$

where Δv is defined by (16). The expression (24) for the s-coordinate of the rightmost point on l completes the proof. □

As to the property (P4) we have the following proposition.

PROPOSITION 7.2. *Property* (P4) *holds if W satisfies the condition*

$$W_{ik} \neq 0 \Rightarrow v_i^R \leqq v_k^R. \tag{31}$$

Proof. This condition on W implies that $\sigma(i, k) \leqq 0$ in (28). Then

$$s + \sigma(i, k) \leqq s \leqq \delta^*(A)$$

and therefore $\delta^*(A') \leqq \delta^*(A)$. □

The statement above says in effect that W should be in a triangular form with respect to the orderings of rows and columns determined by the potentials on the rows of $D(x)$.

The last property (P5) is now considered. First note that the potentials $\{v_i^R, v_j^C\}$ for $G^*(D)$ can be chosen to be integers; then s_{ij} and $\sigma(i, k)$ are also integers. On the other hand, p can be fractional even if $A(t, x)$ contains only integer powers of t and x, i.e., even if

$$A_{ijrs} \neq 0 \Rightarrow r \in \mathbf{Z} \tag{32}$$

holds. Thus, the integrality is not readily guaranteed for $\rho(i, k)$. The following is a preliminary for Proposition 7.4, which gives a sufficient condition for the integrality of $\rho(i, k)$. A sufficient condition for (P5) will be given in Proposition 7.5.

PROPOSITION 7.3. *Assume* (32). *Then*

$$r_{ij} \in \mathbf{Z} \quad \text{if } D_{ij}^* \neq 0.$$

Proof. By (18), D_{ij}^* is the coefficient of the term $x^{s_{ij}} = x^{v_i^R - v_j^C}$ in $D_{ij}(x)$ of (22). The latter is equal, by (22), to the coefficient $A_{ijr_{ij}s_{ij}}$ of the term $t^{r_{ij}}x^{s_{ij}}$ in $A_{ij}(t, x)$ since $(r, s_{ij}) \in E_{ij}$ implies $r = r_{ij}$. That is,

$$D_{ij}^* = A_{ijr_{ij}s_{ij}}.$$

Note that this statement is true whether or not D_{ij}^* vanishes. Hence if $D_{ij}^* \neq 0$, then (32) guarantees $r_{ij} \in \mathbf{Z}$. □

We will introduce an equivalence relation on the row set R, and show in Proposition 7.4 that $\rho(i, k) \in \mathbf{Z}$ under (32) if i and k belong to the same equivalence class. Let us say that $i \in R$ and $k \in R$ are D^*-*adjacent* if $D_{ij}^* D_{kj}^* \neq 0$ for some $j \in C$. We also say that $i \in R$ and $k \in R$ are D^*-*connected* if there exists a chain of indices $i_1, i_2, \cdots, i_m \in R$ such that $i_1 = i$, $i_m = k$, and i_j and i_{j+1} are D^*-connected for $j = 1, \cdots, m-1$. Evidently, D^*-connectedness is an equivalence relation.

PROPOSITION 7.4. *Assume* (32). *Then*

$$\rho(i, k) \in \mathbf{Z} \quad \text{if } i \text{ and } k \text{ are } D^*\text{-connected.}$$

Proof. First suppose i and k are D^*-adjacent. Then there exists $j \in C$ such that $D_{ij}^* D_{kj}^* \neq 0$. Applying Proposition 7.3 we obtain $r_{ij} \in \mathbf{Z}$ and $r_{kj} \in \mathbf{Z}$, from which follows $\rho(i, k) = r_{ij} - r_{kj} \in \mathbf{Z}$.

For a general D^*-connected pair (i, k), relations such as

$$\rho(i, k) = \rho(i, m) + \rho(m, k)$$

prove the claim. □

We have arrived at a condition on W that guarantees the property (P5) as follows.

PROPOSITION 7.5. *Property* (P5) *holds if W satisfies the condition*

$$W_{ik} \neq 0 \Rightarrow i \text{ and } k \text{ are } D^*\text{-connected.} \tag{33}$$

Proof. The proof is obvious from (28) and Proposition 7.4. □

In order to meet the requirements (P1)–(P5) we have imposed three conditions on W, namely,

(27): $$\text{term-rank}\,(WD^*) < n,$$

(31): $$W_{ik} \neq 0 \Rightarrow v_i^R \leqq v_k^R,$$

(33): $$W_{ik} \neq 0 \Rightarrow i \text{ and } k \text{ are } D^*\text{-connected.}$$

Note that these conditions, respectively, imply (P3), (P4), (P5), without mutual dependence. It is important that such W can always be found.

We conclude this section by suggesting a concrete choice of W that meets these three conditions. Since rank $D^* < n$, there exists a nonzero vector $w = (w_i \mid i \in R)$ such that

$$w^T D^* = 0. \tag{34}$$

Let w be such a vector with minimal support, i.e., such that

$$\text{supp } w \equiv \{i \in R \mid w_i \neq 0\}$$

is minimal with respect to set inclusion. Such w can be computed by a Gaussian elimination on D^* with column pivoting. Let $i_0 \in R$ be such that

$$v_{i_0}^R = \min \{v_i^R \mid i \in \text{supp } w\}.$$

The suggested choice of W is

$$W_{ik} = \begin{cases} w_k & \text{if } i = i_0, \\ \delta_{ik} & \text{otherwise,} \end{cases} \tag{35}$$

where δ_{ik} denotes the Kronecker delta, being equal to one or zero accordingly as $i = k$ or not. This W satisfies the above three conditions: (27) follows from (34), (31) from the choice of i_0, and finally (33) from the minimality of supp w. Note also that $\det W = 1$.

Example 7.1. Recall from Example 6.1 that the line l defined by $r = s$ tightly supports $\hat{N}(A)$ for the matrix A of (14), $P = (2, 2) \in l \cap N(A)$ and the rightmost point $P' = (3, 3)$ of $l \cap \hat{N}(A)$ does not belong to $N(A)$. We may take $w = (1, -1, 0)^T$ in (34) for D^* of (26); w has the minimal support, supp $w = \{1, 2\}$. Since $v_1^R = 0$, $v_2^R = 1$, we have $i_0 = 1$, and

$$W = \begin{pmatrix} 1 & -1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

in (35). Then, according to (29), the matrix A is modified to

$$A'(t, x) = \text{diag}\,(1, tx, t^{-2}x^{-2}) \cdot W \cdot \text{diag}\,(1, t^{-1}x^{-1}, t^2x^2) \cdot A(t, x).$$

The resulting matrix A' agrees with the matrix A of (15) in Example 4.1, and $\hat{N}(A')$ is shown in Fig. 4. Note that (P1)-(P5) are satisfied. In particular, l supports $\hat{N}(A')$, the rightmost point of $l \cap \hat{N}(A')$ is $P=(2,2)$, lying to the left of $P'=(3,3)$, and $\delta^*(A')=\delta^*(A)=3$.

8. Starting procedure with artificial terms. This section addresses the problem of finding the initial point in Step 1 of the algorithm when the southwest point of $\hat{N}(A)$ is spurious (cf. Subproblem 1 in § 4). As suggested by the starting procedure of "artificial variables" in the (two-phase) simplex method for linear programming (see, e.g., [5], [8]), we modify the matrix A by introducing a number of "artificial terms" along the diagonal. The modified problem has the genuine southwest point and therefore the algorithm can be started. After a number of steps, we will find that the artificial terms play no roles, and then we are solving the original problem.

Let M be a minimum assignment in $G(A; p)$ for $p \approx -\infty$, and let

$$u_i^R(p) = r_i^R - ps_i^R \quad (i \in R), \qquad u_j^C(p) = r_j^C - ps_j^C \quad (j \in C)$$

be the associated potentials satisfying (19). It should be clear that for sufficiently small p the potentials are linear functions in p and that the coefficients (i.e., r_i^R, s_i^R, r_j^C, s_j^C) can be computed easily once M is found.

Define $A'(t, x) = (A'_{ij}(t, x))$ by

$$A'_{ij}(t, x) = \sum_s \sum_r A_{ijrs} t^{r+r_i^R-r_j^C} x^{s+s_i^R-s_j^C}, \tag{36}$$

i.e.,

$$A'(t, x) = \mathrm{diag}\,(t; r^R) \cdot \mathrm{diag}\,(x; s^R) \cdot A(t, x) \cdot \mathrm{diag}\,(x; -s^C) \cdot \mathrm{diag}\,(t; -r^C).$$

As remarked in § 5, this transformation causes only a coordinate shift in the (s, r)-plane. Since $p \approx -\infty$, (19) means

$$\begin{aligned} &s + s_i^R - s_j^C \geqq 0, \\ &r + r_i^R - r_j^C \geqq 0 \quad \text{if } s + s_i^R - s_j^C = 0. \end{aligned} \tag{37}$$

Therefore,

$$(0, 0) \in \hat{N}(A') \subseteq \{(s, r) \mid s > 0\} \cup \{(s, r) \mid s = 0,\ r \geqq 0\}.$$

Note that $(0, 0) = SW(\hat{N}(A'))$, which corresponds to $(s^*, r^*) = SW(\hat{N}(A))$.

By (37) we may substitute $x=0$ in $A'(t, x)$ and then put $t=0$ to obtain

$$D^* = A'(t, x)\big|_{x=0}\big|_{t=0}. \tag{38}$$

Note that this notation is consistent with (18) when $D(x)$ is defined by (22) with A replaced by A'. Then Proposition 6.3 shows that D^* is singular since the southwest point of $\hat{N}(A)$ is assumed to be spurious.

The artificial terms are introduced into A' as follows. Consider a nonsingular submatrix $D^*[I, J]$ of D^* with row set $I \subset R$ and column set $J \subset C$; then

$$\nu \equiv n - |I| \geqq n - \mathrm{rank}\, D^*.$$

Fixing an arbitrary one-to-one correspondence $\pi : R - I \to C - J$, we define $B(t, x) = (B_{ij}(t, x))$ by

$$B_{ij}(t, x) = \begin{cases} A'_{ij}(t, x) + \alpha t^q x^{-1} & \text{if } j = \pi(i),\ i \in R - I, \\ A'_{ij}(t, x) & \text{otherwise,} \end{cases} \tag{39}$$

using ν artificial terms $\alpha t^q x^{-1}$, where $\alpha \neq 0$ and q is a sufficiently large parameter. To be more specific (cf. Proposition 8.1 below), it suffices to take q such that

$$q > p'_{\max}(s'_{\max}+1)+r'_{\max},$$

where

$$s'_{\max} = n \max\{s \mid A'_{ijrs} \neq 0\}, \qquad p'_{\max} = r'_{\max} - r'_{\min},$$
$$r'_{\max} = n \max\{r \mid A'_{ijrs} \neq 0\}, \qquad r'_{\min} = n \min\{r \mid A'_{ijrs} \neq 0\}.$$

In actual computations, however, α and q are treated as symbols rather than numerical values. It would be natural to choose ν as small as possible, i.e., $\nu = n - \text{rank } D^*$.

We have the following proposition, which shows that the algorithm of § 4 can start for $B(t, x)$ with the initial point $P = (-\nu, \nu q)$ and the essential portions of $N(A')$ and $\hat{N}(A')$ are kept unchanged.

PROPOSITION 8.1. *Let q be sufficiently large, say,*

$$q > p'_{\max}(s'_{\max}+2).$$

(1) $(-\nu, \nu q) \in N(B)$.

(2) $(-\nu, \nu q) = SW(\hat{N}(B))$; *in particular,* $\hat{N}(B) \subseteq \{(s, r) \mid s > -\nu\} \cup \{(s, r) \mid s = -\nu, r \geqq \nu q\}$.

(3) $\hat{N}(B) \subseteq \{(s, r) \mid r + qs \geqq 0\}$.

(4) $N(B) \supseteq N(A')$, $\hat{N}(B) \supseteq \hat{N}(A')$.

(5) $N(B) \cap \{(s, r) \mid r < r'_{\max}\} = N(A')$, $\hat{N}(B) \cap \{(s, r) \mid r < r'_{\max}\} = \hat{N}(A')$.

(6) *A tightly supporting line for $N(A')$ tightly supports $N(B)$.*

(7) *If $\nu = n - \text{rank } D^*$, then $N(B) \cap \{(s, r) \mid r + qs = 0\} = \{(-\nu, \nu q)\}$.*

Proof. (1) The coefficient of $(t^q x^{-1})^\nu$ in $\det B$ is equal to $\pm\alpha^\nu$ multiplied by $\det D^*[I, J]$ $(\neq 0)$.

(2), (3) Consider a perfect matching in $G(B)$ that contains k $(\geqq 0)$ edges corresponding to artificial terms. In the determinant expansion of B this matching yields a term

$$(t^q x^{-1})^k \cdot t^\rho x^\sigma = t^{kq+\rho} x^{-k+\sigma},$$

in which $0 \leqq \sigma \leqq s'_{\max}$, $r'_{\min} \leqq \rho \leqq r'_{\max}$, and $\rho \geqq 0$ if $\sigma = 0$ by (37). Putting $s = -k + \sigma$ and $r = kq + \rho$, and noting $k \leqq \nu$, we see that $s \geqq -\nu$, and that $r \geqq \nu q$ if $s = -\nu$, establishing (2). To show (3) first note that

$$q + \rho \geqq q + r'_{\min} \geqq p'_{\max} s'_{\max} + p'_{\max} + r'_{\max} \geqq r'_{\max}.$$

Then (3) follows from $r + qs = \rho + q\sigma \geqq 0$, since $\rho + q\sigma = \rho \geqq 0$ if $\sigma = 0$ and $\rho + q\sigma \geqq \rho + q \geqq r'_{\max} \geqq 0$ if $\sigma \neq 0$.

(4), (5) If $k \geqq 1$, then $r = kq + \rho \geqq q + \rho \geqq r'_{\max}$, where the last inequality is shown above.

(6) Let l be a tightly supporting line of $N(A')$ passing through two distinct points, say (s_i, r_i) $(i = 1, 2)$, of $N(A')$. The slope p of l is equal to $(r_2 - r_1)/(s_2 - s_1)$, which implies that $|p| \leqq p'_{\max}$. The point $(s, r) = (-k + \sigma, kq + \rho)$ with $k \geqq 1$ lies above l: $r = p(s - s_1) + r_1$ since $(kq + \rho) - p(-k + \sigma - s_1) - r_1 > 0$ follows from $k(q + p) \geqq k(q - p'_{\max}) \geqq q - p'_{\max} > p'_{\max}(s'_{\max} + 1)$, and $-p(\sigma - s_1) \geqq -p'_{\max} s'_{\max}$.

(7) As we have seen in establishing (3), $r + qs = 0$ only if $\sigma = \rho = 0$. Noting that a submatrix of D^* of size greater than $n - \nu$ is singular, we complete the proof by a similar argument to (1). □

Example 8.1. Consider a matrix

$$A(t,x)=\begin{pmatrix} tx+x^2 & t^3 & x \\ tx & 1+tx^2 & 0 \\ t^3 & -x^3 & t^2 \end{pmatrix},$$

for which we have

$$\det A=-t^6x+t^2x^2+t^3x^4-tx^5+\mathrm{spur}\,(t^3x,\,t^4x^3),$$

where, as before, spur $(\cdots)$ is the list of cancelled terms. The combinatorial Newton diagram $\hat{N}(A)$ is shown in Fig. 6, in which the genuine points are indicated by solid disks (●) and the spurious points by open circles (○).

For $p\approx-\infty$ we have

$$u_1^R(p)=0,\quad u_2^R(p)=0,\quad u_3^R(p)=-2-p,$$
$$u_1^C(p)=1-p,\quad u_2^C(p)=0,\quad u_3^C(p)=-p$$

as potentials for $G(A;p)$. According to (36) and (38) we obtain

$$A'(t,x)=\begin{pmatrix} 1+t^{-1}x & t^3 & 1 \\ 1 & 1+tx^2 & 0 \\ 1 & -t^{-2}x^4 & 1 \end{pmatrix}$$

and

$$D^*=\begin{pmatrix} 1 & 0 & 1 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix},$$

where $\mathrm{rank}\,D^*=\mathrm{rank}\,D^*[I,J]=2$ with $I=\{2,3\}$, $J=\{2,3\}$. This shows that $(0,0)=SW(\hat{N}(A'))$ is spurious. Then, introducing $\nu=1$ artificial term we modify A' to

$$B(t,x)=\begin{pmatrix} 1+t^{-1}x+\alpha t^q x^{-1} & t^3 & 1 \\ 1 & 1+tx^2 & 0 \\ 1 & -t^{-2}x^4 & 1 \end{pmatrix}.$$

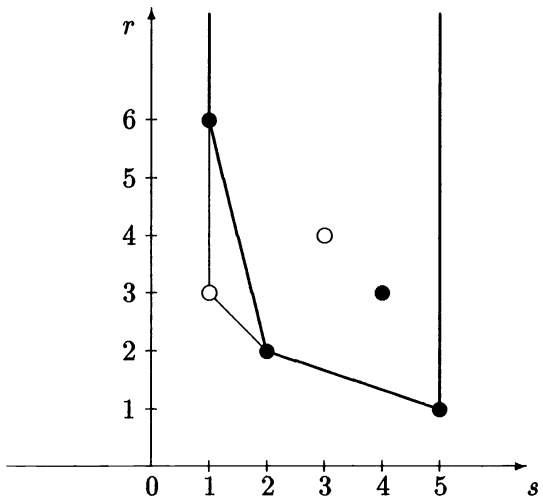


FIG. 6. *Combinatorial Newton diagram* $\hat{N}(A)$ *of Example* 8.1.

Figure 7 illustrates $\hat{N}(B)$, where the genuine points due to the artificial term are indicated by solid triangles (▲).

9. Description of algorithm. In this section we give a complete description of the proposed algorithm outlined in § 4. The algorithm determines all possible first-order approximations $x \sim \gamma t^{-p}$ to the solution of $\det A(t, x) = 0$ for $A(t, x)$ of (1). The statements in brackets $[\cdots]$ are comments for readability, not needed in implementation. Recall that the bipartite graphs $G(A; p)$ and $G^*(D)$ are defined in §§ 3 and 6, respectively. The finite termination and the complexity of the algorithm are considered in the next section.

ALGORITHM.

Step 1 [initial point]

(1) Find a minimum assignment and potentials

$$u_i^R(p) = r_i^R - ps_i^R \ (i \in R), \qquad u_j^C(p) = r_j^C - ps_j^C \ (j \in C)$$

for $G(A; p)$ with $p \approx -\infty$.

(2) $A_{ij}(t, x) := \sum_s \sum_r A_{ijrs} t^{r+r_i^R-r_j^C} x^{s+s_i^R-s_j^C}$ $\quad (i \in R, j \in C);$

$[A_{i,j,r+r_i^R-r_j^C,s+s_i^R-s_j^C} := A_{ijrs}$, cf. (36)]

$D^* := A(t, x)|_{x=0}|_{t=0}$; [cf. (38)]

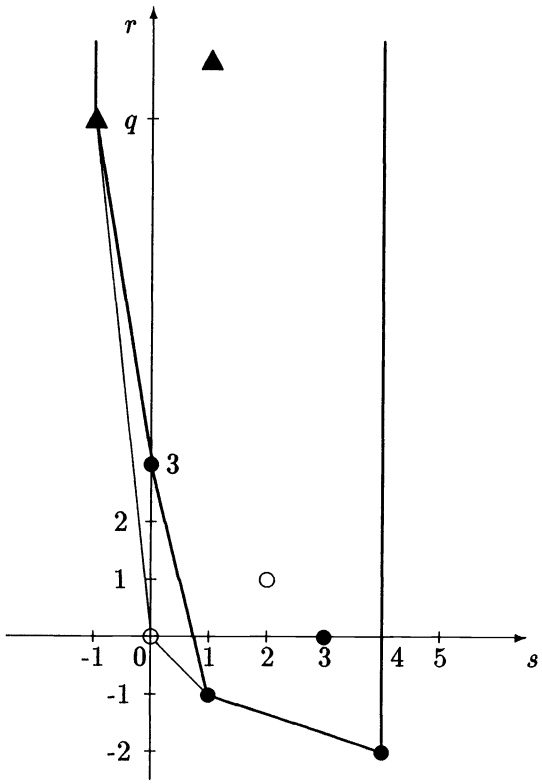


FIG. 7. *Combinatorial Newton diagram* $\hat{N}(B)$ *of Example* 8.1. (▲: *points of artificial terms.*)

Find a maximal nonsingular submatrix $D^*[I, J]$;
$\nu := n - |I|$. [$\nu = n - \text{rank } D^*$]

(3) If $\nu \neq 0$, then

$$A_{ij}(t, x) := \begin{cases} A_{ij}(t, x) + \alpha t^q x^{-1} & \text{if } j = \pi(i),\ i \in R - I, \\ A_{ij}(t, x) & \text{otherwise} \end{cases}$$

where $\pi : R - I \to C - J$ is a one-to-one correspondence. [cf. (39)]

(4) Find a minimum assignment M for $G(A; p)$ with $p \approx -\infty$;
[$P = (s(M), r(M)) = SW(\hat{N}(A)) \in N(A)$]

$$r_{\max} := n \cdot \max \{r \mid A_{ijrs} \neq 0\},$$

$$r_{\min} := n \cdot \min \{r \mid A_{ijrs} \neq 0\},$$

$$p_{\max} := r_{\max} - r_{\min}.$$ [upper bound on p]

Step 2 [solution to relaxation]
[M is a minimum assignment in $G(A; p)$; $P = (s(M), r(M)) \in N(A)$]

(1) Solve a parametric assignment problem on $G(A)$ to find

$$p := \max \{p' \mid M \text{ is a minimum assignment in } G(A; p')\};$$
[p is nondecreasing]

If $p > p_{\max}$ then stop;
[Line l through P with slope p tightly supports $\hat{N}(A)$; $P' :=$ the rightmost point in $l \cap \hat{N}(A)$]
Let u_i^R $(i \in R)$, u_j^C $(j \in C)$ be potentials for $G(A; p)$.

(2) $D_{ij}(x) := \sum \{A_{ijrs} x^s \mid r - ps + u_i^R - u_j^C = 0\}$ $(i \in R, j \in C)$;
[cf. (22)]

Find a maximum assignment M' and potentials v_i^R $(i \in R)$, v_j^C $(j \in C)$ for $G^*(D)$;
[$\Delta v = s$-coordinate of P'; cf. (16), (24)]

$D_{ij}^* :=$ coefficient of $x^{v_i^R - v_j^C}$ in $D_{ij}(x)$ $(i \in R, j \in C)$. [cf. (18)]

Step 3.1 [genuine P'] If $\det D^* \neq 0$, do the following.

(1) Find all solutions $x = \gamma$ $(\neq 0)$ to $\det D(x) = 0$ and output $x \sim \gamma t^{-p}$, unless $p \approx -q$. [$p \approx -q$ arises from artificial terms]

(2) $M := M'$ (with the understanding that $E(G^*(D)) \subseteq E(G(A))$).
[$P := P'$]

(3) Go to Step 2.

Step 3.2 [spurious P'] If $\det D^* = 0$, do the following.

(1) Find w with minimal support such that $w^T D^* = 0$; [cf. (34)]

$$A_{ij}(t, x) := \begin{cases} \sum_k t^{\rho(i_0, k)} x^{\sigma(i_0, k)} w_k A_{kj}(t, x) & \text{if } i = i_0, j \in C, \\ A_{ij}(t, x) & \text{otherwise} \end{cases}$$

where $v_{i_0}^R = \min \{v_i^R \mid w_i \neq 0\}$,

$$\sigma(i_0, k) = v_{i_0}^R - v_k^R,\ \rho(i_0, k) = p\sigma(i_0, k) - (u_{i_0}^R - u_k^R).$$
[cf. (28), (35)]

(2) Go to Step 2.

Remark 9.1. For the parametric assignment problem there are a number of efficient algorithms available. First note that the assignment problem on $G(A; p)$ with a fixed p can be solved by first finding $\min \{r - ps \mid A_{ijrs} \neq 0\}$ for each (i, j) and then applying an algorithm for the assignment problem in a bipartite graph without parallel edges.

Thus a minimum assignment in $G(A; p)$ can be found in $O(|E(G)|)+T_0(n)$ time, where $T_0(n)$ is $O(n^3)$ or $O(n^{2.5})$.

The general scheme of Eisner and Severance [13] for linear parametric optimizations can be specialized to our problem on $G(A; p)$. It finds all breakpoints in a specified open interval $(p^{(1)}, p^{(2)})$ of p by solving the (nonparametric) assignment problem at most $2b+2$ times, where b is the number of breakpoints in this interval. Let us assume the minimum assignments $M^{(1)}$ and $M^{(2)}$, respectively, for $p=p^{(1)}+\varepsilon$ and $p^{(2)}-\varepsilon$ are known (ε denotes a small positive number). A procedure is defined for $(p^{(1)}, M^{(1)}, p^{(2)}, M^{(2)})$ as follows:

(1) If $s(M^{(1)})=s(M^{(2)})$, then there is no breakpoint; otherwise let p' be such that

$$r(M^{(1)})-p's(M^{(1)})=r(M^{(2)})-p's(M^{(2)})\ (\equiv c').$$

(2) Find a minimum assignment M' for $p=p'$.

(3) *If* $c(M')=c'$ then p' is the unique breakpoint in $(p^{(1)}, p^{(2)})$; otherwise apply this procedure recursively to $(p^{(1)}, M^{(1)}, p', M')$ and $(p', M', p^{(2)}, M^{(2)})$.

In Step 2(1) we need to find the next larger breakpoint, and not all the breakpoints. The general scheme of Gusfield [14], when specialized to our problem on $G(A; p)$, gives a polynomial time algorithm for this task, although its time complexity is roughly the square of that for finding a minimum assignment in $G(A; p)$ with a particular value of p.

Finally, we mention the parametric network simplex method. The assignment problem can be formulated as a linear programming problem, and the general method known as parametric simplex method is applicable with additional advantage from the underlying network structure (cf. [5], [8], [10]). In view of the anticycling and antistalling schemes due to Cunningham [6], [7], this approach seems promising from theoretical and practical points of view. Note that the pivots to find the next larger breakpoint are degenerate.

Remark 9.2. In case no artificial terms are introduced in Step 1(3), the minimum assignment M in Step 1(4) is essentially the same as the one, say $M^{(1)}$, found in Step 1(1). The change of A in Step 1(2) does not affect the minimality of $M^{(1)}$, although, by our notational convention, the labels of the edges of $G(A; p)$ are changed from $(ijrs)$ to $(i, j, r+r_i^R-r_j^C, s+s_i^R-s_j^C)$, i.e., formally,

$$M=\{(i, j, r+r_i^R-r_j^C, s+s_i^R-s_j^C)\,|\,(ijrs)\in M^{(1)}\}.$$

Remark 9.3. In Step 3.1(1) the nonzero solutions to $\det D(x)=0$ are to be found. The lowest and the highest degree of a nonzero term in $\det D(x)$ are given by the s-coordinates of P and P', respectively. It is more convenient to work with

$$\bar{D}(y)=\operatorname{diag}(y; v^R)\cdot D(1/y)\cdot\operatorname{diag}(y; -v^C)$$

since each entry of $\bar{D}(y)$ is a polynomial in y and $\det \bar{D}(0)=\det D^*\neq 0$. The degree of $F(y)=\det \bar{D}(y)$ is known as

$$\deg_y F(y)=\Delta v-s(M).$$

Note this is equal to the number of nonzero solutions to $\det D(x)=0$ if the underlying field is the complex number field $\mathbf{C}$, since $F(0)\neq 0$. In some applications, the numerical values of γ are not needed, but only the number of solutions corresponding to p is of interest; this is given by $\Delta v-s(M)$. In other applications, the numerical solutions of $F(y)=0$ are to be found either by symbolic-algebraic methods or by purely numerical methods. In the latter case, simultaneous approximation-type methods (such as those ascribed to Aberth, Durand, Ehrlich, Kerner, and Weierstrass [1], [12], [17], [22]) would be suitable if the underlying field is $\mathbf{C}$; note that the degree of F is known.

Remark 9.4. In Step 3.2(1) we have adopted the choice (35) of W for concreteness. Other choices are also possible.

Remark 9.5. As explained in § 2.1 the higher-order terms in the Puiseux-series solution (4) are determined from (7). We can compute the second-order term following $\gamma_1 t^{-p_1}$ by applying the above algorithm to $A'(t, x) = A(t, t^{-p_1}(\gamma_1 + x))$. Note, however, that $p < 0$ for higher order terms. The southwest point $SW(\hat{N}(A'))$ is (almost) always spurious and the artificial terms are needed.

10. Termination and complexity. In this section we consider the termination and the complexity of the proposed algorithm. In the first place we consider the probabilistic behavior of the algorithm. As already noted in § 3 (cf. Proposition 3.1 in particular), $\hat{N}(A)$ differs from $N(A)$ only because of accidental numerical cancellation. Let us fix the structure (i.e., the graph $G(A)$) of the input matrix $A(t, x) = A^{(0)}(t, x) = (A^{(0)}_{ij}(t, x))$ and regard the numerical values of nonzero coefficients A_{ijrs} as real- (or complex-) valued independent random variables with continuous distributions. Then we have $\hat{N}(A) = N(A)$ with probability one, which means that all the exponents p of $x \sim \gamma t^{-p}$ can be determined by finding all the breakpoints for $G(A; p)$ without any modification of the matrix A; in particular, Step 3.2 of our algorithm is not performed at all. Hence we obtain the following statement.

PROPOSITION 10.1. *The average time complexity of the algorithm per exponent p is bounded by a polynomial in n, except for the numerical determination of the coefficients* γ *in Step* 3.1.

We put

$$E_0 = |\{(ijrs) \,|\, A^{(0)}_{ijrs} \neq 0\}| \ (= |E(G(A^{(0)}))|),$$

$$L_0 = \max \{|s|, |r| \,|\, A^{(0)}_{ijrs} \neq 0\}.$$

We denote by r_{den} the least common multiple of the denominators of all r with $A^{(0)}_{ijrs} \neq 0$.

Let $A^{(1)}(t, x) = (A^{(1)}_{ij}(t, x))$ denote the matrix A at the end of Step 1. In general this matrix contains ν artificial terms introduced in Step 1(3). To avoid inessential complication in presentation, we treat the case where no artificial terms are involved. Then it would be easy to see that the main results (Proposition 10.2(2), Proposition 10.3) remain true in the general case.

Define

$$s_{\max} = n \cdot \max \{s \,|\, A^{(1)}_{ijrs} \neq 0\};$$

we assume $s_{\max} \geqq 2$; also recall $r_{\max}$ and $r_{\min}$ in Step 1(4). If we define E_1 and L_1 for $A^{(1)}$ similarly to E_0 and L_0 with $A^{(0)}$ replaced by $A^{(1)}$, we see from (10) that

$$E_1 = E_0, \quad L_1 \leqq 2nL_0, \quad \max \{s_{\max}, r_{\max}, |r_{\min}|\} \leqq nL_1.$$

Obviously,

$$N(A^{(1)}) \subseteq \hat{N}(A^{(1)}) \subseteq \{(s, r) \,|\, 0 \leqq s \leqq s_{\max}, \ r_{\min} \leqq r \leqq r_{\max}\}.$$

Let us consider the matrix A in the loop of Steps 2, 3.1, and 3.2. Since $N(A)$ is invariant, the above inclusion implies

$$N(A) \subseteq \{(s, r) \,|\, 0 \leqq s \leqq s_{\max}, \ r_{\min} \leqq r \leqq r_{\max}\}. \tag{40}$$

This shows that the slope p of a tightly supporting line of $N(A)$ is not greater than $p_{\max}$ defined in Step 1(4). Hence the stopping criterion in Step 2(1).

By the definition of r_{den}, the input matrix $A^{(0)}(t, x)$ involves only integer powers of $\tau = t^{1/r_{\mathrm{den}}}$ and x. We may further assume the integrality of potentials whenever the

costs are integers (see § 2.2). This implies that $A^{(1)}$ also involves only integer powers of τ and x. Then it follows from (P5) in § 7 that this property is inherited by all A, and hence

$$\hat{N}(A) \subseteq \{(s, r) \mid s \in \mathbf{Z},\ r_{\mathrm{den}} r \in \mathbf{Z}\}.$$

On the other hand, (P4) in § 7 guarantees that $\delta^*(A) \leqq \delta^*(A^{(1)})$, which implies

$$\hat{N}(A) \subseteq \{(s, r) \mid s \leqq s_{\max}\}.$$

In Step 2(2) the rightmost point $P' = (s', r')$ of $l \cap \hat{N}(A)$ is computed implicitly. Namely, since the line l with slope p passes through $P = (s(M), r(M)) = (s_0, r_0)$ and P', we have

$$s' = \Delta v, \qquad r' = p(s' - s_0) - r_0,$$

where Δv is defined by (16).

The following guarantees the finite termination of the algorithm for a general input matrix with fractional powers of t and gives a pseudopolynomial (i.e., polynomial in n and L_0) bound on the number of steps in the whole algorithm for an input matrix with integer powers of t and x. Note that r_{den} may not be pseudopolynomially bounded.

PROPOSITION 10.2. (1) *When no artificial terms need be introduced, the points P' produced by the algorithm are all distinct and belong to*

$$\mathcal{P} = \{(s, r) \mid s \in \mathbf{Z},\ r_{\mathrm{den}} r \in \mathbf{Z},\ 0 \leqq s \leqq s_{\max},\ r_{\min} s \leqq r \leqq p_{\max} s\}.$$

Hence the number of executions of Step 2 is bounded by

$$|\mathcal{P}| \leqq (s_{\max} + 1)^2 (p_{\max} - r_{\min} + 1) r_{\mathrm{den}} \leqq (2n^2 L_0 + 1)^2 (6n^2 L_0 + 1) r_{\mathrm{den}}.$$

(2) *If the given matrix $A = A^{(0)}$ contains only integer powers of t and x, the number of executions of Step 2 is pseudopolynomially bounded by the input size.* (This statement does not preclude the case with artificial terms.)

Proof. (1) Let us define a stage to be a series of executions of the loop of Steps 2 and 3.2 without interruption by Step 3.1. During a stage $P = (s_0, r_0)$ is not changed; $0 \leqq s_0 \leqq s_{\max}$, $r_{\min} \leqq r_0 \leqq r_{\max}$ since $P \in N(A)$. Let $P_1 = (s_1, r_1)$ be the extreme point, if any, of $N(A)$ with next larger s-coordinate; P' should coincide with P_1 at the end of the stage. We define $p_1 = (r_1 - r_0)/(s_1 - s_0)$, or $p_1 = p_{\max}$ if no such P_1 exists.

The line l rotates around P with nondecreasing slope p; while p is kept unchanged, s' decreases at least by one at each execution. Hence the points P' produced in this stage are all distinct and belong to

$$\{(s, r) \mid s \in \mathbf{Z},\ r_{\mathrm{den}} r \in \mathbf{Z},\ p_0(s - s_0) + r_0 < r \leqq p_1(s - s_0) + r_0,\ s \leqq s_{\max}\} \subseteq \mathcal{P},$$

where p_0 is the value of p at the beginning of the stage. This expression shows that the points P' are all distinct during the whole algorithm. A simple calculation shows the bound on $|\mathcal{P}|$.

(2) The claim follows from (1) if no artificial terms are introduced. The proof for the general case is similar; note that q can be chosen to be pseudopolynomially large, as shown in Proposition 8.1. □

Finally we will show, for theoretical completeness, that the proposed algorithm can be implemented so that its running time has a pseudopolynomial worst-case bound if the input matrix $A^{(0)}(t, x)$ involves only integer powers of t and x. As stated in Proposition 10.2, the number of iterations is pseudopolynomially bounded. The problem to be considered is that the transformation of the matrix $A(t, x)$ in Step 3.2 may cause an indefinite increase of the number of edges $|E(G(A))|$ in $G(A)$, which is equal

to the total number of nonzero terms in $A(t, x)$. Recall that the complexity for finding the next larger breakpoint for $G(A)$ is bounded by a polynomial in n and $|E(G(A))|$.

To establish a pseudopolynomial bound, we will show that it suffices to keep only a pseudopolynomial number of nonzero terms in $A(t, x)$. To be more precise, we will consider the case without artificial terms and show that, in that case, we may retain only those terms $A_{ijrs}t^r x^s$ with

$$-s_{\max} \leqq s \leqq s_{\max}/n \tag{41}$$

and

$$0 \leqq r - ps + u_i^R - u_j^C \leqq \beta_{\max}, \tag{42}$$

where

$$\beta_{\max} = r_{\max} + 2p_{\max}s_{\max} - r_{\min}(s_{\max} + 1).$$

Note that the number of such terms is bounded by a polynomial in n, E_0, L_0, and r_{den}.

Recall that, for each term $A_{ijrs}t^r x^s$ of $A(t, x)$, or for each edge $(ijrs)$ of $G(A)$, in Step 2, we have

$$s \leqq \delta^*(A) \leqq \delta^*(A^{(1)}) = s_{\max}/n \tag{43}$$

and (19):

$$r - ps + u_i^R - u_j^C \geqq 0.$$

Consider a particular term $A_{ijrs}t^r x^s$ of $A(t, x)$ and denote by $A'(t, x)$ the matrix with this term deleted from $A(t, x)$. Let M be a perfect matching in $G(A)$ which contains the edge $(ijrs)$ corresponding to this term.

First suppose that $s < -s_{\max}$, then $s(M) < 0$ by (43). This shows $\hat{N}(A) - \hat{N}(A') \subseteq \{(s, r) \mid s < 0\}$, from which we see in view of (40) that the deletion of this term does not affect the subsequent behavior of the algorithm. (Note, however, that $N(A')$ would be different from $N(A)$; e.g., the point $(s(M), r(M))$, which is spurious for A, may possibly become genuine for A'.) Therefore, we may throw away those terms which do not lie in the range of (41).

Next suppose that

$$r - ps + u_i^R - u_j^C > \beta_{\max}.$$

It then follows from (19) that

$$r(M) - ps(M) + \Delta u > \beta_{\max},$$

where Δu is defined by (21). Since line l (with equation $r = ps - \Delta u$) passes through $P = (s_0, r_0)$ with $0 \leqq s_0 \leqq s_{\max}$, $r_{\min} \leqq r_0 \leqq r_{\max}$, we have

$$\Delta u = ps_0 - r_0 \leqq p_{\max}s_{\max} - r_{\min}.$$

Combining these two and using (41) and $r_{\min} \leqq p \leqq p_{\max}$, we obtain

$$r(M) - p_{\max}s(M) > (p - p_{\max})s(M) - \Delta u + \beta_{\max} \geqq r_{\max}.$$

This shows

$$\hat{N}(A) - \hat{N}(A') \subseteq \{(s, r) \mid r - p_{\max}s > r_{\max}\},$$

which implies, as before, that the deletion of this term does not affect the subsequent behavior of the algorithm. Thus we have shown that we may also throw away those terms which do not lie in the range of (42).

The above arguments are readily extended to the general case with artificial terms to establish the pseudopolynomiality of the proposed algorithm in case r_{den} is polynomially bounded by n, E_0, and L_0; note that q can be chosen to be pseudopolynomially large (cf. Proposition 8.1).

PROPOSITION 10.3. *The proposed algorithm can be implemented to run in time polynomial in n, E_0, L_0, and r_{den}, except for the numerical determination of the coefficients γ in Step* 3.1.

11. Conclusion. Implementation details and practical efficiency of the proposed algorithm will be reported later.

The author is thankful to Bill Cunningham and Dan Gusfield for indicating the relevant references [13] and [14] concerning the parametric assignment problem, and to Takao Asano, Masaaki Sugihara, Mike Trick, and the anonymous referee for helpful comments.

Appendix. Notation.

- $A(t, x) = (A_{ij}(t, x))$: $A_{ij}(t, x) = \sum_{s \in \mathbf{Z}} \sum_{r \in \mathbf{Q}} A_{ijrs} t^r x^s$ (§ 1, (1))
- $\alpha t^q x^{-1}$: artificial term (§ 8, (39))
- $B(t, x) = (B_{ij}(t, x))$: matrix with artificial terms (§ 8, (39))
- C: column set of matrices (§ 3)
- c_e: cost for edge e in G (in general) (§ 2.2)
- $c_{ij} = \max\{s \mid D_{ijs} \neq 0\} = \deg_x D_{ij}(x)$: cost in $G^*(D)$ (§ 6.1)
- $\tilde{c}_{ij} = c_{ij} - v_i^R + v_j^C$ (17)
- $c_{ijrs}(p) = r - ps$: cost in $G(A; p)$ (§ 3)
- $\tilde{c}_{ijrs} = r - ps + u_i^R - u_j^C$ (19)
- $c(M) = \sum_{e \in M} c_e$: cost of matching M (in general) (§ 2.2)
- $D(x) = (D_{ij}(x))$: $D_{ij}(x) = \sum_{s \in \mathbf{Z}} D_{ijs} x^s$ (§ 6.1)
  $D(x) = (D_{ij}(x))$: $D_{ij}(x) = \sum \{A_{ijrs} x^s \mid (s, r) \in E_{ij}\}$ (22)
- $\tilde{D}(x) = (\tilde{D}_{ij}(x))$: $\tilde{D}_{ij}(x) = \begin{cases} D_{ijc_{ij}} x^{c_{ij}} & \text{if } \tilde{c}_{ij} = 0, \\ 0 & \text{otherwise} \end{cases}$ (§ 6.1)
- $D^* = (D^*_{ij})$: $D^*_{ij} = \begin{cases} D_{ijc_{ij}} & \text{if } \tilde{c}_{ij} = 0, \\ 0 & \text{otherwise} \end{cases}$ ((18), (38))
- $\delta(D) = \deg_x \det D(x)$ (§ 6.1)
- $\hat{\delta}(D)$: maximum cost of an assignment in $G^*(D)$ (§ 6.1)
- $\delta^*(A) = \max\{\deg_x A_{ij}(t, x) \mid i \in R, j \in C\}$ (§ 7)
- $\Delta u = \sum_i u_i^R - \sum_j u_j^C$ (21)
- $\Delta v = \sum_i v_i^R - \sum_j v_j^C$ (16)
- $\partial^+ e$: initial vertex of edge e (§ 2.2)
- $\partial^- e$: terminal vertex of edge e (§ 2.2)
- $E_0 = |\{(ijrs) \mid A^{(0)}_{ijrs} \neq 0\}|$ $(= |E(G(A^{(0)}))|)$ (§ 10)
- $E_1 = |\{(ijrs) \mid A^{(1)}_{ijrs} \neq 0\}|$ $(= |E(G(A^{(1)}))|)$ (§ 10)
- $E_{ij} = \{(s, r) \mid \tilde{c}_{ijrs} = 0, (ijrs) \in E(G)\}$ (23)
- $f(t, x) = \det A(t, x) = \sum_s \sum_r f_{rs} t^r x^s$ (§ 1, (2))
- $G = G(A) = G(A; p)$: bipartite graph for A (with multiple edges) (§ 3)
- $G^* = G^*(D)$: (simple) bipartite graph for D (§ 6.1)
- γ_i: coefficients in (4) (§ 2.1)
- l: supporting line of (combinatorial) Newton diagram (§§ 4, 9)
- $L_0 = \max\{|s|, |r| \mid A^{(0)}_{ijrs} \neq 0\}$ (§ 10)
- $L_1 = \max\{|s|, |r| \mid A^{(1)}_{ijrs} \neq 0\}$ (§ 10)

- M: (perfect) matching (§ 2.2)
- n: size of matrices (§ 1, (1))
- $N(A)=\{(s, r)\,|\,f_{rs}\neq 0\}$: Newton diagram for A (§ 3, (2))
- $\hat{N}(A)=\{(s(M), r(M))\,|\,M$: perfect matching in $G(A)\}$: combinatorial Newton diagram for A (§ 3, (12))
- $\nu=n-|I|$: (often) number of artificial terms (§ 8)
- p: exponent p_1 in (4), slope of l (§ 4)
- p_i: exponents in (4); $p_i<0$ for $i\geqq 2$ (§ 2.1)
- $p_{\max}=r_{\max}-r_{\min}$ (§ 9)
- $p'_{\max}=r'_{\max}-r'_{\min}$ (§ 8)
- $P'=(s', r')$: rightmost point of $l\cap\hat{N}(A)$ (§§ 4, 6.2)
- $\mathcal{P}$: region for P' (§ 10)
- π: one-to-one correspondence $R-I\to C-J$ (§ 8)
- R: row set of matrices (§ 3)
- $r^*=\min\{r\,|\,(s^*, r)\in N\}$: r-coordinate of southwest point (5)
- $r_{ij}=ps_{ij}-u_i^R+u_j^C$ (§ 7)
- $r(s)=\min\{r\,|\,f_{rs}\neq 0\}$ (§§ 1, 2)
- r_{den}: least common multiple of denominators of r (§ 10)
- $r_{\max}=n\cdot\max\{r\,|\,A_{ijrs}\neq 0\}$ (§ 9)
- $r'_{\max}=n\cdot\max\{r\,|\,A'_{ijrs}\neq 0\}$ (§ 8)
- $r_{\min}=n\cdot\min\{r\,|\,A_{ijrs}\neq 0\}$ (§ 9)
- $r'_{\min}=n\cdot\min\{r\,|\,A'_{ijrs}\neq 0\}$ (§ 8)
- $\rho(i, k)=r_{ij}-r_{kj}=p(v_i^R-v_k^R)-(u_i^R-u_k^R)$ (§ 7)
- $s^*=\min\{s\,|\,(s, r)\in N\}$: s-coordinate of southwest point (5)
- $s_{ij}=v_i^R-v_j^C$ (§ 7)
- $s_{\max}=n\cdot\max\{s\,|\,A^{(1)}_{ijrs}\neq 0\}$ (§ 10)
- $s'_{\max}=n\cdot\max\{s\,|\,A'_{ijrs}\neq 0\}$ (§ 8)
- $SW(N)=(s^*, r^*)$: southwest point (5)
- $\sigma(i, k)=s_{ij}-s_{kj}=v_i^R-v_k^R$ (§ 7)
- u_i^R, u_j^C: potentials for minimization (in $G(A; p)$) $u_i^R(p)=r_i^R-ps_i^R$, $u_j^C(p)=r_j^C-ps_j^C$ (§ 2.2, (8), § 6.2)
- v_i^R, v_j^C: potentials for maximization (in $G^*(D)$) (§ 2.2, (9), § 6.1)
- w: vector in transformation matrix W (34)
- W: transformation matrix (§ 7, (35))
- $x(t)=\gamma_1 t^{-p_1}+\gamma_2 t^{-p_1-p_2}+\cdots$ (4)

REFERENCES

[1] O. ABERTH, *Iteration methods for finding all zeros of a polynomial simultaneously*, Math. Comput., 27 (1973), pp. 339-344.

[2] Y. AKIZUKI, Y. NAKAI, AND M. NAGATA, *Algebraic Geometry*, Iwanami, Tokyo, 1987. (In Japanese.)

[3] B. BUCHBERGER, G. E. COLLINS, AND R. LOOS, *Computer Algebra—Symbolic and Algebraic Computation*, Computing Supplementum 4, Springer-Verlag, Berlin, New York, 1982.

[4] E. BRIESKORN AND H. KNÖRRER, *Plane Algebraic Curves*, Birkhäuser, Boston, 1986.

[5] V. CHVÁTAL, *Linear Programming*, W. H. Freeman, San Francisco, CA, 1983.

[6] W. H. CUNNINGHAM, *A network simplex method*, Math. Programming, 11 (1976), pp. 105-116.

[7] ———, *Theoretical properties of the network simplex method*, Math. Oper. Res., 4 (1979), pp. 196-208.

[8] G. B. DANTZIG, *Linear Programming and Extensions*, Princeton University Press, Princeton, NJ, 1963.

[9] J. DAVENPORT, Y. SIRET, AND E. TOURNIER, *Computer Algebra—Systems and Algorithms for Algebraic Manipulation*, Academic Press, New York, 1988.

[10] U. DERIG, *Programming in Networks and Graphs*, Lecture Notes in Economics and Mathematical Systems, Vol. 300, Springer-Verlag, Berlin, New York, 1988.

[11] J. DIEUDONNE, *Calcul infinitesimal*, Hermann, Paris, 1968.
[12] E. DURAND, *Solution numérique des equations algébriques*, Vol. 1, Masson, Paris, 1968.
[13] M. J. EISNER AND D. G. SEVERANCE, *Mathematical techniques for efficient record segmentation in large shared database*, J. Assoc. Comput. Mach., 23 (1976), pp. 619–635.
[14] D. GUSFIELD, *Parametric combinatorial computing and a problem of program module distribution*, J. Assoc. Comput. Mach., 30 (1983), pp. 551–563.
[15] E. HILL, *Analytic Function Theory*, Vol. 2, Chelsea, New York, 1962.
[16] K. IKEDA AND K. MUROTA, *Critical initial imperfection of structures*, Internat. J. Solids and Structures, 26 (1990), pp. 865–886.
[17] I. O. KERNER, *Ein Gesamtschrittverfahren zur Berechnung der Nullstellen von Polynomen*, Numer. Math., 8 (1966), pp. 290–294.
[18] E. L. LAWLER, *Combinatorial Optimization: Networks and Matroids*, Holt, Rinehart and Winston, New York, 1976.
[19] L. LOVÁSZ AND M. PLUMMER, *Matching Theory*, North-Holland, Amsterdam, 1986.
[20] K. MUROTA, *Systems Analysis by Graphs and Matroids—Structural Solvability and Controllability*, Algorithms and Combinatorics, Vol. 3, Springer-Verlag, Berlin, New York, 1987.
[21] ———, *Structure-oriented algorithm for determining dynamical degree*, Report No. 89591-OR, Institute of Econometrics and Operations Research, University of Bonn, Bonn, FRG, 1989.
[22] L. PASQUINI AND D. TRIGIANTE, *A globally convergent method for simultaneously finding polynomial roots*, Math. Comp., 44 (1985), pp. 135–149.
[23] F. P. PREPARATA AND M. I. SHAMOS, *Computational Geometry*, 2nd ed., Springer-Verlag, Berlin, New York, 1988.
[24] T. SASAKI, *Symbolic and Algebraic Manipulation*, Information Processing Society of Japan, Tokyo, 1981. (In Japanese.)
[25] A. SCHRIJVER, *Theory of Linear and Integer Programming*, John Wiley, New York, 1986.
[26] M. M. VAINBERG AND V. A. TRENOGIN, *Theory of Branching of Solutions of Non-linear Equations*, Noordhoff, Groningen, the Netherlands, 1974.